

Java 游戏高级编程

(美) David Wallace Croft 著

彭 晖 译

清华大学出版社

北 京

前言

花代价所换来的一点才智，抵过别人传授的数倍不止。

——本杰明·富兰克林

本书针对的是那些想使用最新的高性能技术，创建面向 Web 和桌面的复杂 2D 计算机游戏的 Java 程序员。本书循序渐进地介绍了一个可重用的动画库，每一章都依赖于前面的一些章节。在需要说明这些主题的地方，都详细地介绍了每一个新库类的源代码，并且给出了实际使用这些类的示例游戏。免费许可使您可以直接修改和调整库代码和示例游戏，开发自己的游戏程序。

目的

本书也可取名为《现代 Java 游戏编程》。在以前出版的 Java 游戏编程的书籍中介绍的很多技术和主题，已被 Java 编程语言和它的高级类库所淘汰。这包括像线程管理、事件管理、图形用户界面、网络通信、持久性和部署等方面。在本书中，我对这些变更作了适当的注释，并简要地对这些技术作了一些对比。

虽然通过将每一帧中需要重绘像素的数量最小化就可以在 Java 中创建高速的动画，但是在最近发布的 Java 编程语言的 1.4 版中引入的新类，又提供了通过可移植方式直接访问视频硬件的方法。正如在本书中介绍的一样，即使每一帧中需要更新的像素很多，也可以实现高速的帧速率。在这里介绍了一个示例程序，用来演示在真彩色高分辨率模式中，成功地将全屏动画和显示器 75Hz 的刷新率同步。

尽管本书最初的设计是面向游戏，但是读者应该知道这里介绍的技术也可以用在其他应用方面，包括广告和仿真。嵌在 Web 页面中的动画调幅广告 applet 会立即闪现在您的大脑中。调幅广告可以使用强大的通用编程语言(例如 Java)编写的这个事实，充分发掘它们超越动画的辅助功能的潜能，包括活动数据连接。在仿真领域，学生和科学家都可以从可视化上获益。当这些模型使用 Java 表示的时候，用户就可以与它们进行交互、更改参数并观察新的结果。

读者对象

本书专门针对具有丰富经验、正准备和渴望将他们独特的游戏设计思想转换为部署实体的 Java 开发人员。对这些读者而言，从想了解有趣的游戏特定逻辑以来，一直在学习 Java 编程语言和实现基本的基础结构代码。

同时, 这些 Java 高级开发人员希望详细地了解可能会影响游戏性能的可重用库中的优化选项。在介绍游戏库和本书示例游戏中的这些源代码时, 我假定读者已经掌握了一些基本的主题, 例如 Java 编程语言、面向对象的编程(Object Oriented Programming, OOP)技术、图形用户界面(Graphical User Interface, GUI)组件库, 以及一些设计模式等。

范围

本书简要(没有深入地)介绍了那些虽然必需但又不是专门用于 Java 游戏编程的技术。这包括通用的开放源代码开发工具和通用的标准以及应用编程接口(API), 这些都是很多经验丰富的 Java 游戏程序员已经使用过的技术。在书中, 还提供了在这些方面需要进一步阅读的参考书目。

本书介绍的很多可重用库都利用了核心 Java 平台里可用的最新 API、当前的 J2SE 1.4 版。本书没有介绍那些过时的 API, 例如 AWT(Abstract Window Toolkit)。相反, 深入地研究了应用现代 API 的具体技术, 包括 Java 游戏编程用于动画设计的 Swing。

除了 Java 2D 和 Image I/O API 以外, Java Media API 中绝大部分 API 都没有在本书中介绍, 因为我考虑到它们在游戏开发中会有一些限制。在游戏中使用 Java 3D、JMF(Java Media Framework)或者 Java Speech API, 就强制玩家下载安装一个可选包和它的本地代码实现。我相信这种附加的步骤将会使很多潜在的玩家退缩, 尤其是那些遇到安装问题和下载延迟问题的玩家。尽管已经对 J2SE 内核库的一部分进行了安装, 但是 Java Sound API 仍然需要一个 soundbank 文件, 这个文件对有些操作系统而言是默认包含在 Java 运行库环境(JRE)发布中的, 而对 Windows 而言则不是。Java 共享数据工具包(Java Shared Data Toolkit, JSDT)API, 尽管也可以在网络游戏中使用, 但是它们好像已经被废止了。我在游戏编程里, 还没有发现确实需要使用 Java 高级图像(Java Advanced Imaging, JAI)的地方。

本书并没有回避将 Java 与平台专用代码混合的 Java 本地接口(JNI)API 的使用, 但是也不鼓励这样使用。本书中给出的所有代码都是用纯粹、可移植的 Java 语言编写的, 它们能够在所有安装有 Java 虚拟机的平台上运行。作为一个开发人员, 我发现不使用定制的本地库也可以实现高性能。作为一个玩家, 我更加喜欢用纯 Java 编写的游戏, 因为我不必因为安全风险而烦恼, 并且我知道游戏将会运行在我喜欢的平台上, 而不管这是个什么平台。

本书没有介绍使用 Java 2 Micro Edition(J2ME)平台进行 Java 游戏开发。我希望像摩尔定律中描述的那样, 硬件性能的快速发展能使 J2ME 很快过时。如今的手持 PDA 使用的微处理器的性能已经与 3 年以前使用的最好计算机的处理器性能一样强大了。现在在 PDA 上安装 J2SE 和运行 Swing 应用程序已经成为可能。如果在近几年内看到 J2EE 运行在手表上, 我将不会感到惊奇。

在学习后面的关于多玩家联网模式的相关章节时, 确实需要读者具有一些 J2EE 的知识。然而, 这也限制使用 Servlet API 进行开发。更为复杂的 J2EE API(例如 Enterprise Java Beans(EJBs))的使用方法, 本书并没有进行介绍。所有示例的网络游戏都会运行在一个简单的 servlet 容器以内, 并且也不需要安装一个完全的应用服务器。

内容简介

在第1章“配置开发环境”中，我向读者介绍了将要在整本书中使用的代码库。描述了这个库的设计，并给出了编译示例游戏的说明。可能会在游戏开发环境中使用的一些附加代码、图形和音频文件，也在本章确定了下来，还介绍了软件开发的一些流行工具。以模板的形式给出了一个演示 Java 游戏编程基础知识的游戏示例。

在第2章“部署框架”中，我们关注的重点是在可以安装在多种不同类型的平台上的一个框架中部署游戏。在动画线程管理技术的相关内容中，介绍了一个被广泛使用的框架接口以及 applet 生命周期。在对适合于游戏的不同发布而进行不同部署的选项进行了一个简要的回顾以后，给出了允许游戏部署在不同开发环境中而不需要任何修改的一个抽象层的源代码示例。

在第3章“Swing 动画”中，介绍了一个基于 Swing 的动画库，这个动画库会在整本书中使用。这一章主要考虑的问题是动画的性能和灵活性。在介绍核心动画引擎类的源代码时简单介绍了优化权衡。

在第4章“动画库”中，介绍了通用动画任务的一个类集合。这些类提供了很多功能，例如场景管理和 Sprite 实现。它们也用来介绍怎样开发与核心动画引擎类互操作的游戏特定代码。

在第5章“高级图形技术”中，介绍了高级图形技术，包括硬件加速图形、多缓冲和全屏独占模式。给出了使这些技术的使用变得更加方便的可重用类和使用这些可重用类的示例游戏。对其中的每一个技术，都介绍了关于它们使用方法的一些限制。

在第6章“持久数据”中，对在各种部署框架中加载和保存游戏数据的机制进行了对比。在这个过程中还介绍了可以用于大多数游戏编程的可重用数据持久性的类库。对更加高级的持久性需求的其他选择也进行了考虑。

在第7章“游戏体系结构”中，介绍了适合于 Java 游戏开发的面向对象的软件体系结构的优点。使用这种体系结构的一个示例游戏是作为新游戏开发模板进行介绍的。本章还简单介绍了一下数据驱动的设计。

在第8章“A*算法”中，介绍了现在在游戏行业中最流行和最常用的一个人工智能(AI)算法的实现。本章还介绍了使用 A*算法绕过障碍物寻找路径的示例。

在第9章“HTTP 隧道技术”中，介绍一个可以在大部分 Java 游戏部署环境里常见的安全限制以内进行操作的一个联网库。本章还给出了一个示例，在这个示例中，数据在服务器和客户机之间进行传递。

在第10章“HTTP 轮询机制”中，将联网库扩展以支持在线多玩家游戏。使用轮询来同步客户机和服务器上的游戏状态。在由单玩家游戏向多玩家联网游戏的转换中，介绍了我所推荐的软件体系结构的灵活性。

在第11章“HTTP Pulling 机制”中，事件驱动的消息是作为轮询的另一个选择而推荐使用的。使用在前面两章中介绍的这些类，演示了这是怎样在无符号 applet 的安全限制以内实现的。使用这些技术进行进一步开发的一些建议已经超出了本书的范围，这里只进行简单的介绍。

本书合作站点

本书应该保持其“高级性”。为此，我专门为这本书建立了一个 Web 站点。从这个站点上，您能够订阅与本书相关的电子邮件列表，包括补充的指南和勘误表。您将能够测试和运行本书所介绍的示例游戏，并下载开放源代码库和用来创建这些游戏的公共领域的多媒体文件。针对老师，我已经在 Creative Commons Attribution License 下发布了我在讲授游戏开发课程中使用的课程提纲、幻灯片和布置给学生的作业。我也提供了我的联系方式，这样您可以将您的建议和意见反馈给我，以便我以后进行更正。

在开始阅读第 1 章以前，请先访问 <http://www.croftsoft.com/library/books/ajgp>，并订阅 *Advanced Java Game Programming* 通知邮件列表。



目 录

第 1 章 配置开发环境	1
1.1 升级到 Java 1.4	1
1.2 内核类	1
1.3 在线运行 Demo 版程序	2
1.4 深入游戏库	3
1.4.1 croftsoft 目录	3
1.4.2 arc 目录	4
1.4.3 bin 目录	4
1.4.4 doc 目录	4
1.4.5 ext 目录	5
1.4.6 lib 目录	5
1.4.7 lic 目录	5
1.4.8 res 目录	5
1.4.9 src 目录	6
1.4.10 tmp 目录	6
1.5 XML 简介	6
1.6 使用 Ant 编译	7
1.7 使用开放源代码	10
1.7.1 学习版权的基本知识	10
1.7.2 选择一个许可	11
1.7.3 重命名修改后的代码	12
1.7.4 共享源代码	13
1.8 为游戏获取多媒体资源	13
1.8.1 图片资源	13
1.8.2 音频资源	14
1.9 基本示例	15
1.9.1 修改源代码	15
1.9.2 修改构建文件	26
1.10 小结	27
1.11 参考文献	27
第 2 章 部署框架	28
2.1 部署为 applet	28
2.1.1 实现生命周期方法	28
2.1.2 管理 applet 动画线程	30

2.1.3	读取 JAR 文件	34
2.1.4	使用插件升级客户端	36
2.1.5	了解存在的限制	38
2.1.6	applet 签名	38
2.1.7	缓存 applet	39
2.2	部署为可执行 JAR	39
2.2.1	生成清单文件	39
2.2.2	对不安全性进行保护	40
2.3	用 Java Web Start 进行部署	40
2.3.1	准备发布文件	41
2.3.2	访问默认浏览器	43
2.3.3	使用反射进行动态链接	46
2.4	将多个 applet 部署为一个 applet	47
2.4.1	MultiAppletStup	48
2.4.2	MultiAppletNews	50
2.4.3	Lifecycle	54
2.4.4	LifecycleWindowListener	54
2.4.5	MultiApplet	58
2.4.6	CroftSoftCollection	65
2.5	小结	67
2.6	参考文献	68
第 3 章	Swing 动画	69
3.1	ComponentAnimator	69
3.1.1	更新和绘制阶段	70
3.1.2	精灵的定义	70
3.1.3	ExampleAnimator	71
3.2	RepaintCollector	73
3.2.1	Swing 串行化	73
3.2.2	动画的问题	73
3.2.3	RepaintCollector	75
3.2.4	SimpleRepaintCollector	75
3.2.5	BooleanRepaintCollector	77
3.2.6	CoalescingRepaintCollector	78
3.2.7	其他实现	81
3.3	LoopGovernor	81
3.3.1	固定的延迟	82
3.3.2	帧速率同步	84
3.3.3	SamplerLoopGovernor	85
3.3.4	WindowedLoopGovernor	85
3.4	AnimatedComponent	90
3.5	小结	98

3.6	参考文献	98
第4章	动画库	99
4.1	ComponentPainter 实现	99
4.1.1	NullComponentPainter	99
4.1.2	ArrayComponentPainter	99
4.1.3	ColorPainter	101
4.1.4	SpacePainter	103
4.1.5	TilePainter	106
4.2	ComponentUpdater 实现	114
4.2.1	NullComponentUpdater	114
4.2.2	ArrayComponentUpdater	115
4.2.3	EdgeScrollUpdater	115
4.3	ComponentAnimator 实现	120
4.3.1	NullComponentAnimator	120
4.3.2	TileAnimator	120
4.3.3	FrameRateAnimator	122
4.3.4	CursorAnimator	126
4.4	Sprite 实现	131
4.4.1	Sprite	131
4.4.2	AbstractSprite	132
4.4.3	IconSprite	135
4.4.4	BounceUpdater	137
4.4.5	IconSequenceUpdater	141
4.5	小结	143
第5章	高级图形技术	144
5.1	硬件加速图像	144
5.1.1	Image	144
5.1.2	BufferedImage	145
5.1.3	VolatileImage	145
5.1.4	自动图像	146
5.1.5	兼容图像	146
5.1.6	缓存算法	149
5.2	多缓冲和多线程	150
5.3	全屏独占模式	160
5.3.1	启用全屏模式	160
5.3.2	FullScreenToggler	161
5.3.3	配置帧	165
5.3.4	改变显示模式	165
5.3.5	DisplayModeLib	166

5.3.6	GraphicsDeviceLib	166
5.3.7	消除锯齿	167
5.3.8	BufferStrategyAnimatedComponent	168
5.3.9	FullScreenDemo	170
5.3.10	使用独占模式的顾虑	177
5.4	小结	177
5.5	参考文献	177
第 6 章	持久数据	178
6.1	数据格式	178
6.1.1	对象串行化	178
6.1.2	属性	181
6.1.3	XML	182
6.1.4	瓦片地图图像	190
6.1.5	随机种子	193
6.2	持久性机制	195
6.2.1	JAR 资源文件	196
6.2.2	用户主目录文件	197
6.2.3	JNLP 持久性	200
6.2.4	applet 持久性	203
6.2.5	稳固持久性	206
6.2.6	嵌入式数据库	211
6.2.7	服务器端的持久性	212
6.3	数据完整性	214
6.3.1	消息摘要	214
6.3.2	散列缓存	215
6.4	小结	217
6.5	参考文献	217
第 7 章	游戏体系结构	218
7.1	接口	218
7.2	继承	218
7.3	目标 Mars	221
7.4	模型-视图-控制器	221
7.4.1	模型	222
7.4.2	多重接口继承	224
7.4.3	视图	234
7.4.4	控制器	240
7.5	复合 MVC	243
7.5.1	复合模型	243
7.5.2	复合视图	254

7.5.3	复合控制器	260
7.5.4	将三者进行组合	260
7.6	数据驱动设计	264
7.6.1	AnimationInit	264
7.6.2	AnimatedApplet	265
7.7	小结	269
7.8	参考文献	269
第 8 章	A*算法	270
8.1	Cartographer	271
8.2	NodeInfo	272
8.3	AStar	273
8.4	AStarTest	279
8.5	SpaceTester	283
8.6	GridCartographer	283
8.7	GradientCartographer	286
8.8	TankConsole	289
8.9	TankOperator	290
8.10	StateSpaceNode	291
8.11	TankCartographer	292
8.12	DefaultTankOperator	293
8.13	PlayerTankOperator	297
8.14	小结	299
8.15	参考文献	300
第 9 章	HTTP 隧道技术	301
9.1	测试示例	302
9.2	可重用的客户端代码	303
9.2.1	Encoder	304
9.2.2	Parser	304
9.2.3	StreamLib	304
9.2.4	StringCoder	305
9.2.5	HttpLib	306
9.2.6	Queue	308
9.2.7	ListQueue	309
9.2.8	Loopable	312
9.2.9	Looper	313
9.2.10	HttpMessagePusher	315
9.3	游戏特定的客户端代码	318
9.4	可重用的服务器端代码	323
9.4.1	Server	323

9.4.2	HttpGatewayServlet	324
9.5	游戏特定的服务器端代码	327
9.6	打包 WAR	330
9.6.1	web.xml	330
9.6.2	build.xml	331
9.7	小结	332
9.8	参考文献	332
第 10 章	HTTP 轮询机制	333
10.1	测试示例程序	333
10.2	可重用的客户端代码	334
10.2.1	SerializableCoder	334
10.2.2	HttpMessagePoller	335
10.2.3	Consumer	340
10.2.4	QueuePuller	340
10.2.5	HttpMessageClient	342
10.3	游戏特定的客户端代码	345
10.3.1	Request	345
10.3.2	AbstractRequest	346
10.3.3	FireRequest	346
10.3.4	MoveRequest	347
10.3.5	ViewRequest	348
10.3.6	GameData	348
10.3.7	Synchronizer	350
10.3.8	NetController	354
10.3.9	NetMain	356
10.4	可重用的服务器端代码	357
10.4.1	XmlBeanCoder	358
10.4.2	SerializableLib	359
10.5	游戏特定的服务器端代码	360
10.5.1	GameInit	360
10.5.2	Player	362
10.5.3	NetGame	363
10.5.4	MarsServer	370
10.5.5	MarsServlet	376
10.6	小结	378
10.7	参考文献	378
第 11 章	HTTP Pulling 机制	379
11.1	测试示例	379
11.2	可重用的客户端代码	381

11.2.1	Authentication	381
11.2.2	Id	381
11.2.3	LongId	382
11.2.4	ModelId	383
11.2.5	SeriModelId	383
11.3	游戏特定的客户端代码	384
11.3.1	Request	384
11.3.2	CoalesceableRequest	384
11.3.3	ChatController	385
11.3.4	Response	386
11.3.5	CreateUserConsumer	386
11.3.6	Event	387
11.3.7	ChatClient	387
11.3.8	ChatSynchronizer	393
11.3.9	ChatPanel	395
11.3.10	ChatApplet	398
11.4	服务器端代码	400
11.4.1	User	400
11.4.2	UserStore	401
11.4.3	PullServer	402
11.4.4	MoveServer	403
11.4.5	SeriChatGame	404
11.4.6	ChatServer	409
11.5	跟踪消息	411
11.5.1	多玩家联网模式	412
11.5.2	单玩家本地模式	413
11.6	扩展示例	414
11.7	小结	414
11.8	参考文献	415
附录 A	源代码索引	416
附录 B	CVS 简介	420
B.1	检出代码	420
B.2	创建自己的项目	421
B.3	并行程序设计	423
B.3.1	实施代码所有权	423
B.3.2	互相监视	424
B.3.3	创建分支	425
B.4	参考文献	425

第 1 章 配置开发环境

身教胜于言传。

——本杰明·富兰克林

本章主要介绍怎样配置开发环境，以及怎样使用本书所介绍的游戏库和示例程序，创建自己的游戏程序。

1.1 升级到 Java 1.4

编译和运行本书的示例游戏，需要下载安装 Java 2 软件开发工具箱(Software Development Kit, SDK)和 1.4 以上版本的 Java 2 标准版(Java 2 Standard Edition, J2SE)。其中的 Java 2 指的是 Java 2 开发平台，而 Java 1.4 指的是 Java 2 开发平台的某一个具体版本。尽管可以去责怪造成这种混乱的 SUN Microsystems 的营销人员，但是事实确实如此。

由于本书的示例游戏代码都使用了 J2SE 1.4 版本中新增的图形加速类，所以只能在 J2SE 1.4 以上版本中运行这些示例代码。J2SE 1.4 版本中新增的那些类极大地提高了 Java 2 的动画性能。在编著本书的时候，J2SE 1.4 版已经可以在多个操作系统平台上运行了，这些操作系统包括 Linux、Mac OS X、Solaris 和 Windows。如果没有 1.4 版以上的 J2SE，可以立即到 <http://java.sun.com> 网站去下载，然后将它安装在包括 Apple 在内的那些常用开发平台上。Apple 的开发人员还可以从 <http://developer.apple.com/java/> 网站下载 Java 的最新版本。

游戏代码应该是完全不需要进行修改就可以直接在支持 J2SE 1.4 版的各种平台上运行。话虽然是这么说的，但是这里还是要提醒您，本人只在 Linux 系统和 Windows 系统上对代码进行了测试。只有根据使用 Mac OS X 和 Solaris 系统的读者对该书的反馈意见，我才能知道这些代码对这两种操作系统的适应情况。

1.2 内核类

J2SE 定义了一个标准的应用编程接口(API)库，这个标准库中包含数千个类，这些类又被集合到内核包中。在每一个 J2SE 的兼容系统中，这些内核类肯定都是预安装的。这样可以使开发人员可以更加轻松，因为它减少了客户机需要下载和安装的代码量。Java.lang 就是这样的一个内核包，该内核包包含了绝大部分 Java 平台所需要的基本类。

在目标平台上，非内核类一般都不是预安装的，因此，在使用这些非内核类的时候，需要进行慎重地考虑。这些非内核类一般都称为可选包(optional package)。最近也才将它们通称为标准扩展(standard extension)包。之所以说它们是“扩展”的，是由于它们不是和内核类库一起发布的；之所以说它们是“标准”的，是由于它们为与其底层实现无关的特殊功能定义了标准的接口。javax.media.j3d 就是一个可选包，它是一个专门用于渲染 3D 场景画面的类库。

由于内核包的名称都是以前缀 Java 开头，而标准扩展包的名称都是以前缀 Javax 开头，所以我们很容易区分哪一个包是内核包，哪一个包是标准扩展包。但是，这是不久以前的情况，而现在在内核类库中也包含有以非 java(例如 javax.swing 包)开头的包。我不清楚 Sun Microsystems 为什么会改变这种习惯的命名方法，但是我猜测这与最近微软的官司可能会有些关系。我知道在这种更改决定做出以后，总会不断地迷惑一些甚至是高水平的 Sun 工程师。在某一些具体的问题上，他们可能将必须检查整个 Java 库。也许他们会将这个 Java 称为 Java 3 平台的 2.0 版。如果他们要是真的这样做，我还是希望他们能够考虑将包的命名规则恢复为它原来的那种习惯约定。

大部分示例游戏代码都只依赖于内核类。这保证了它们在 J2SE 平台上不再需要进行另外的安装就能够运行。然而，也有一些游戏还要依赖于可选包。在介绍这些示例游戏的时候，我会提示您。

J2SE 用于较小平台(例如嵌入式设备)的小型版是 Java 2 Micro Edition(J2ME)。J2SE 用于那些繁重任务平台(例如应用服务器)的大型版是 Java 2 Enterprise Edition(J2EE)。本书不包括 J2ME 的内容，对 J2EE 的内容也只涉及到其冰山一角。在需要使用 J2EE 代码的地方，我事先会提示您；在需要使用可选包的时候，您只能和我采取一样的方法。

1.3 在线运行 Demo 版程序

本书 Web 站点的 URL 是 <http://www.croftsoft.com/library/books/ajgp/>。在该站点上，可以通过预编译和在线安装测试 Demo 版程序，以便对有些内容首先有个亲身的体验。我还会不断地更新这个 Demo。请访问本书的 Web 站点以获取其最新版本的内容，也请订阅邮件列表以获取更新的通告邮件。

这个 Demo 包含有几个 Java 动画程序，其中包括游戏程序和模拟仿真程序。可以从左到右单击屏幕顶部的标签，以浏览整个列表。其中最有趣的一个程序就是 Sprite(精灵)，这是一个“精灵”动画测试程序。如图 1-1 所示，可以更改选项和参数，观察动画执行的不同效果。



图 1-1 Sprite 程序的 Demo 版

1.4 深入游戏库

可以从本书的站点上以压缩归档文件的形式，为游戏库下载源代码、图形和音频文件的快照(snapshot)。该代码快照中包含本书归档的版本号。还可以通过 CVS 客户端，下载这些代码的最新版。其他一些更详细的信息，请参见附录 B。

下载并解压缩以后，就可以使用这些源代码编译和运行示例游戏。下面几节会详细介绍源代码库的目录结构。

1.4.1 croftsoft 目录

该归档文件内部的顶层目录是 `croftsoft/`。该目录中包含主要的构建文件(build file)、`readme.txt` 文件和游戏库的子目录。下面列表详细列举了该游戏库子目录的结构。有些子目录并没有包含在归档文件(ZIP 文档)中，但是，在编译示例游戏的时候，会自动创建它们。

- `arc/`——归档包
- `bin/`——二进制文件和实用程序
- `doc/`——javadoc 文档
- `ext/`——扩展库
- `lib/`——编译后的类库
- `lic/`——许可
- `res/`——资源文件
- `src/`——源代码
- `tmp/`——临时目录

1.4.2 arc 目录

作为构建过程的一部分，在为了进行发布而对代码打包时，可执行的 Java 归档文件(JAR)和 Web 应用程序归档文件(WAR)都应该存放在这个 arc 目录中。第 2 章将更加详细地介绍创建 JAR 文件的过程。

1.4.3 bin 目录

子目录 bin/是存放批处理文件、脚本和实用程序的地方。按照我的理解，bin 就是二进制 binaries 的简写。这是一个存放与源代码文本文件相对应的、编译过的二进制实用程序的地方。关于 bin 目录名称的起源问题，我的测猜可能也不准确：它可能就是简单地代表 bin，就只是一个存放东西的地方。

```
path=%path%;C:\home\croft\cvs\croftsoft\bin
```

应该将上面这段代码中的这个目录追加到环境变量 path 中。这个目录中包含有一个批处理文件 javainit.bat，该文件的作用是初始化开发环境。在刚开始进行开发的时候，这是需要运行的一个非常有用的批处理文件。

```
subst J: C:\home\croft\cvs\croftsoft
subst K: C:\home\croft\cvs\croftsoft\src\com\croftsoft\apps
```

在使用 Windows 操作系统进行开发的时候，在工作目录中，我使用这个命令代替驱动器符号 J:。这样，其他的批处理文件和构建文件就与这些工作目录的变更独立开来，因为它们使用的路径是 J:\src 而不是 C:\home\croft\cvs\croftsoft\src。将驱动器符号 K:作为应用程序源代码目录的快捷方式——这里也是我完成大部分游戏开发工作的地方。

也可以使用 javainit.bat 来初始化在开发过程中可能需要使用的那些环境变量。在本人的机器上，我定义了很多供应用程序和实用程序使用的环境变量，这些环境变量都带有 JAVA_HOME、J2EE_HOME 和 JBOSS_HOME 等这样的目录名称。注意，这时在编译和运行示例程序时，不再需要定义任何环境变量。

```
javac -deprecation -d J:\lib -sourcepath J:\src -classpath J:\lib;[...] %1
```

目录 bin/中也包含有批处理文件 jc.bat，该文件执行上面的这条编译命令。这里省略了其中的一些类的路径值，以使该命令尽量简短。%1 是这个命令行的第一个参数，它表示要进行编译的 Java 源代码文件的名称。如果想编译当前目录中的全部文件，那么就可以键入这样的命令：jc *.java。

1.4.4 doc 目录

目录 doc/中存放源代码的 javadoc 文档文件。不应该在这个地方存放手动创建项目的文档文件，因为偶尔可能会想清空这个目录，以删除那些已被删除类的 javadoc 文档，这样就不用担心可能会删除一些无法用其他文件替代的重要文档。因为 javadoc 实用程序会从源代码中自动生成该目录的内容，所以不要直接编辑这些文件或将这些文件添加到版本控制中。

1.4.5 ext 目录

目录 `ext/` 存放构建时需要使用的可选包和扩展库的 JAR 文件。在 JAR 文件以前，开发人员都使用 `zip` 文件对类进行归档，因此偶尔也会在这个目录中看到以 `.zip` 为扩展名的文件。

```
-classpath J:\lib;J:\ext\j2ee.jar;J:\ext\javaws.jar;[...]
```

我喜欢将扩展名为 JAR 的文件都汇集到这个目录，而不简单地在最初安装它们的地方直接使用它们的一个主要原因就是，这个目录能使类的路径变得更具有可读性。我当然更加愿意使用一个短参数(例如 `J:\ext\javaws`)来代替像 `C:\Program Files\Java WebStart\javaws.jar` 这样的长参数。另外，那些目录名称中带有空格的目录(例如目录 `Program Files`)，有时会混淆在类路径中使用这种目录的应用程序。

最后也是最重要的就是，它完善了项目所使用的整个扩展库。在项目中，可以很容易地将它们(可选包和扩展库)放入到版本控制之中。由于通常只在需要时才从第三方站点上下载副本，因此将第三方二进制文件加入版本控制之中似乎有些奇怪。但是，这仍然是不可靠的，因为代码中可能需要使用较早的版本库，而这些较早的版本库在供应商的站点上又不再进行发布了。将 `ext/` 目录放在版本控制之中以后，我们就有了各种不同扩展库(明确地和项目一起可靠运行的扩展库)版本的一个快照。这也使同伴的工作可以变得更加轻松；如果需要转移到另外的一台机器上进行开发，这甚至还可以使这种迁移工作变得更加容易。因为我们可以从版本控制系统中拖出 JAR 文件，而不需要从很多个不同供应商的 Web 站点上重新进行下载。

1.4.6 lib 目录

目录 `lib/` 是已编译的 Java 类(这些文件的扩展名均为 `.class`)的目标目录。这个路径是 `javac` 编译命令的 `-classpath` 参数，也是 `java` 执行命令的 `-cp` 参数。

`lib` 是“库(library)”的缩写。但是，要知道，这个目录仅仅是为 `src/` 目录下的源代码文件中已编译的那一部分准备的。我们应该在其他目录(例如 `ext/` 目录)中保存第三方代码库，因为我们有时可能需要清空 `lib/` 目录，以删除已编译类的过期版本，这样就不用担心会删除一些很重要且无法替代的东西。由于 `lib/` 目录中的所有文件都是从 `src/` 目录中产生的，所有就没有必要将它们纳入到版本控制之中。

1.4.7 lic 目录

目录 `lic/` 中存储开放源代码软件许可的副本。本章稍后会对此进行更加详细的介绍。

1.4.8 res 目录

我们通常使用目录 `res/` 来保存完成构建所需要的那些资源文件(除了 Java 源代码文件以外的其他资源文件)。这些资源文件包括 JAR 清单、WAR 配置文件、多媒体资源(例如音频文件、图形文件)和静态数据文件。

在 `src/` 目录层次结构的内部，根据其包前缀，每一个源代码文件都有一个适当的存储位置。这一点与在构建中用到的其他类型的文件不同。与其将这些文件都存放在 `src/` 或其下级子目录

中(例如, 与包含应用程序主类的包相对应的目录), 还不如将它们集中存储在一个单独的目录中, 这个目录就是 `res/` 目录。

1.4.9 src 目录

目录 `src/` 中存放的是还没有编译的 Java 源代码: 带有 `.java` 文件扩展名的那些文件。这个目录也是存放超文本标记语言(Hypertext Markup Language, HTML)的 `javadoc` 描述的地方。`javadoc` 实用程序在源代码路径中的每一个目录中搜索 `package.html` 文件, 并假定这个文件中包含有与目录层次相对应的包中的类的描述。

1.4.10 tmp 目录

目录 `tmp` 是在构建的过程中创建的、用于临时保存工作文件的一个目录。一般在构建工作结束的时候, 这个目录就会被自动销毁。如果由于编译错误, 构建过程被中途中断, 那么这个目录就不会被自动删除。如果这个目录存在也不会带来什么危害, 因为下一次成功的构建首先就会自动地将它删除, 然后再创建一个, 最后再将其删除。如果您决定手动地删除这个目录, 惟一的限制就是不能在构建过程正在进行的时候删除它。

1.5 XML 简介

为了构建、打包和部署游戏, 还需要理解扩展标记语言(Extensible Markup Language, XML)的基本知识。XML 在存储游戏数据、存储硬件驱动上的用户参数以及通过网络传递多个玩家信息等方面是非常有用的。这一节将对 XML 进行简要介绍。

XML 允许我们使用人类可读的纯文本来描述数据。在下面的例子中, 请注意每一个数据值都是怎样封装到由一个元素的开始标记和一个结束标记组成的标记对中的。标记的名称就是数据字段的名称。

```
<book>
  <title>Advanced Java Game Programming</title>
  <author>David Wallace Croft</author>
  <publisher>Apress</publisher>
  <pubdate>2004</pubdate>
</book>
```

那些熟悉 HTML 的人会很快就感到二者之间的相似性。除了那些我自己创建的元素名称(例如 `book` 和 `title`)以外, 它看起来非常像 HTML。HTML 中已经定义了数据元素 `title`, 但是在此, 我给 `title` 赋予了新的语义, 换句话说, 也就是在我的 `book` 定义的上下文中, `title` 的意思发生了改变。实际上, 之所以将 XML 称为“可扩展”的语言, 正是由于这种能够定义新数据元素的能力。

尽管 XML 在定义新数据元素名称的能力上具有很大的灵活性, 但是在 XML 格式中还是有足够多的限制, 这些限制使解析器不需要为每一个新的定义进行专门的定制。这些限制包含一个严格分层的元素嵌套。例如下面的带有重叠标记边界的 XML 语法, 就会产生解析错误。

```
<b><i>Illegal XML</b></i>
```


虽然上面的这段代码在 HTML 中可能也是正确的,但是在扩展超文本标记语言(Extensible Hypertext Markup Language, XHTML)——XML 的另一种代替 HTML 的定义中,嵌套元素的正确方法应该如下所示。

```
<i><b>Legal XML</b></i>
```

现在,成百甚至上千个 XML 的定义几乎涵盖了从家谱到电子商务的所有领域。只要数据需要以透明的方式使用标准解析器(在各种主要编程语言里都可以方便地使用)进行交换,XML 都对其提供了解决方案。

有关 XML 主题的深层信息是很容易从大量的源代码中获得的,因为 XML 已经迅速地成了人们广泛采用的技术。就易学易用的参考书来说,我推荐使用由 Robert Eckstein 编写的 *XML Pocket Reference* 的第 2 版(O'Reilly & Associates, 2001)。

1.6 使用 Ant 编译

Ant 是开放源代码(Open Source)的工具。我们可以使用这个开放源代码工具编译示例游戏的源代码。Ant 的功能比以前早期版本工具(例如 make)的功能更加强大,因为它是可扩展和跨平台的。可以考虑将 Ant 作为一个通用的脚本语言来代替平台专用的批处理文件或脚本文件。如果它缺少所需要的某一个功能,Ant 乐意帮助您集成所有想编写的以提高其性能的 Java 代码。

Ant 最初是在 Apache Jakarta Project 网站上发布的,它很快就在 Java 世界里获得了普遍的认同。如果您还没有学会怎样使用 Ant,应该考虑去学习使用。可以从阅读 <http://ant.apache.org/> 网站的在线文档来学习 Ant。

```
<project name="myproject" default="compile">
  <target name="compile">
    [...]
  </target>
  <target name="archive" depends="compile">
    [...]
  </target>
</project>
```

Ant 自己带有关于怎样从一个 XML 构建文件(带有一个默认的名称 build.xml)中编译源代码的说明。一个构建文件是作为一个带有多个 target(目标)的项目进行组织的。每一个目标中含有 0 个或多个 task(任务),这些任务就是由 Ant 处理器执行的命令。

某一个目标中的任务通常都是在可能依赖于前面一个或多个目标的成功完成的一个集合中相互关联的。例如,可能有一个名为 archive 的目标,这个目标的任务就是归档最新版本的源代码,并将它转移到备份目录中。archive 目标可能要依赖于名为 compile 的另一个目标,这个 compile 目标的任务就是编译整个代码库。如果在带有目标 archive(指定为命令行参数)的项目构建文件上运行 Ant, Ant 就会首先启动 compile 目标。如果这个步骤失败了,它就会终止操作,而不会启动 archive。

通过在库的根目录中的 build.xml 文件上使用默认的目标运行 Ant,就可以编译并运行绝大部分游戏源代码。下面逐行列出了用于构建主示例程序文件的第一部分构建代码。可以将这些代码改为对自己的游戏进行编译和打包的构建代码。

```
<project name="croftsoft" default="demo">
```

如果没有以命令行参数的形式指定目标就执行了这个构建文件,那么 Ant 就使用默认的 demo 目标。

```
<property name="arc_dir" value="arc"/>
<property name="res_dir" value="res"/>
<property name="src_dir" value="src"/>
<property name="tmp_dir" value="tmp"/>
```

在构建文件的其余部分,为了防止目录名称变为硬编码(hard-coded)的名称,可以在此使用属性变量来定义它们。一旦定义了这些属性值,在构建文件中就再也不能重新定义它们了。您可能需要调整这些目录的名称,以适合于自己的习惯。

```
<target name="init">
  <mkdir dir="${arc_dir}"/>
  <delete dir="${tmp_dir}"/>
  <mkdir dir="${tmp_dir}"/>
  <tstamp>
    <format property="TODAY_ISO" pattern="yyyy-MM-dd"/>
    <format property="YEAR" pattern="yyyy"/>
  </tstamp>
</target>
```

默认的目标 demo 间接地依赖于目标 init,因此,这里还要再回顾一下目标 init。init 目标首先创建输出和临时工作目录。如果输出目录 arc_dir 已经存在,构建文件将会接受这个目录(不会重新创建这个目录)而不会引发异常。临时目录会被重新创建,以确保不会将以前构建中的原文件遗留在这个目录中。

任务 tstamp 将属性 TODAY_ISO 设置为当前日期的值,这个日期的格式是国际标准化组织(International Standards Organization, ISO)格式(例如 1999-12-31)。下面的有些目标就使用这个属性对文件名称进行归档。

```
<target name="shooter_prep" depends="init">
  <copy todir="${tmp_dir}/media/shooter">
    <fileset dir="${res_dir}/apps/shooter/media">
      <include name="shooter_bang.png"/>
      <include name="shooter_boom.png"/>
      <include name="shooter_rest.png"/>
      <include name="bang.wav"/>
      <include name="explode.wav"/>
    </fileset>
  </copy>
</target>
```

在准备编译 demo 的过程中,游戏的源代码文件被从资源目录复制到临时工作目录。上面的代码演示了复制命令的一个变种形式,该变种使用了一个 fileset 标记。与其为每一个文件都使用一个单独的 copy 命令,还不如使用 fileset 标记在一个命令中将所有文件都复制到临时工作目录。也可以在 fileset 标记中使用通配符和模式匹配来描述所要包含的文件。我习惯逐一通过名称指定文件,因为这样就能够准确地知道正在将什么内容打包。


```

<target name="collection_prep1"
  depends="basics_prep,dodger_prep,fraction_prep,mars_prep"/>

<target name="collection_prep2"
  depends="road_prep,shooter_prep,sprite_prep,tile_prep,zombie_prep"/>

<target name="collection_prepare"
  depends="collection_prep1,collection_prep2">

  <copy file="${res_dir}/apps/images/croftsoft.png"
    todir="${tmp_dir}/images"/>

</target>

```

目标 `collection_prepare` 要依赖于目标 `collection_prep1` 和 `collection_prep2`。这样将其分开就是为了让 `depends` 标记值不会运行太久。`collection_prepare` 的目的是将加入 CroftSoft Collection 的每个游戏的资源文件都复制到临时工作目录。它也会复制一些其他的资源文件，例如 `croftsoft.png`，这是一个用作帧图标文件。

```

<target name="collection_compile" depends="collection_prepare">

  <javac srcdir="${src_dir}" destdir="${tmp_dir}">
    <include
      name="com/croftsoft/apps/collection/CroftSoftCollection.java"/>
    <include name="com/croftsoft/ajgp/basics/BasicsExample.java"/>
    [...]
    <include name="com/croftsoft/apps/zombie/Zombie.java"/>
  </javac>

</target>

```

可以使用 `javac` 任务来编译 `src_dir` 目录中的源代码，并将编译过的类文件输出到 `tmp_dir` 中。一般情况下，只需要包含主类，`javac` 将自动编译所需要的所有其他类。但是，在这种情况下，每一个游戏应用小程序(applet)都是动态进行链接而不是静态进行链接的，我们还需要显式地对它们进行命名。第 2 章将对此进行进一步的介绍。

```

<target name="collection_jar" depends="collection_compile">
  <echo
    file="manifest.txt" message=
    "Main-Class: com.croftsoft.apps.collection.CroftSoftCollection" />
  <jar
    basedir="${tmp_dir}"
    destfile="${arc_dir}/collection.jar"
    manifest="manifest.txt"
    update="false"/>
  <delete file="manifest.txt"/>
  <delete dir="${tmp_dir}" />
</target>

<target name="demo" depends="collection_jar">
  <java fork="true" jar="${arc_dir}/collection.jar"/>
</target>

```

如果目标 `collection_compile` 编译成功, `collection_jar` 目标就会得到执行。`collection_jar` 目标将临时工作目录中的内容进行打包, 这个临时工作目录中既包含编译过的代码, 也包含资源文件。在输出目录 `arc_dir` 中, 如果原来的 JAR 文件已经存在, `collection_jar` 就会重写这个 JAR 文件。完成以后, 临时目录将会被自动删除, 新创建的可执行 JAR 文件就会在一个单独的 Java 虚拟机(Java Virtual Machine, JVM)上启动, 以准备进行测试。

构建文件的其余部分还含有一些附加目标, 这些附加目标用于编译那些需要使用 J2SE 库所包含的内核库以外的可选包的示例。还在构建文件中嵌入了注释的一个快速预览, 这些注释可以告诉您需要使用哪些可选包来编译和运行在默认的 `demo` 目标中没有包含的一些附加示例游戏。现在, 您可以完全放心地忽略这些附加的目标, 因为在本书的后续章节中还会介绍它们。

```
ant basics
```

正确地安装了 Ant 以后, 只要在包含 `build.xml` 文件的目录中输入命令 `Ant`, 默认的 `demo` 目标就会编译并运行。如果想使用另外一个目标, 只需要在命令行中以参数的形式给出目标就行了, 正如上面的示例代码一样。例如目标 `basics`, 就可以编译并启动在本章最后给出的那个示例游戏。如果您还没有这样做, 请返回到本章的前面, 使用 Ant 和示例构建文件来重新构建和运行这个 `demo`。

1.7 使用开放源代码

到此为止, 您应该已经能够成功地编译和运行示例程序了。您可以准备学习怎样修改和结合本书所介绍的代码库创建自己的游戏。但是, 在准备作更多的努力之前, 首先应该知道一些使用限制和使用要求。

1.7.1 学习版权的基本知识

在阅读了很多关于版权的书籍以前, 版权对我而言一直是一个神秘的东西。下面是我觉得开发人员必须要去了解的、有关版权的一些很关键的基本知识(译者注: 这里说的是美国版权法)。

- 不需要在作品上另加上一个版权声明, 也不需要为作品取得版权而专门做一些文书工作。通过简单的创造就可以自动获得版权。如果作品由于版权受到侵害需要获得赔偿时, 在作品上加上版权声明和注册版权确实也有一定帮助。
- 不需要在版权声明中加上 `All rights reserved`。现在法律已经作了更改, 这一声明现在是默认的了。
- 如果您只是一个雇员, 默认情况下, 老板就是您在工作中创作作品的版权所有者。如果您是一个独立的签约人, 默认情况下(除非您的合约中专门作了另外的说明), 您就是版权的所有者。如果不是您自己草拟合约的话, 您的合约总是会有另外的陈述。
- 如果您和其他人一起创作了一个作品, 那么如果没有其他的协议约定的话, 你们将共同拥有该作品的版权。使用和发行时, 需要你们达成一致同意, 你们必须共享一个作品的收益。

- 签订一个作品的版权归共同拥有，但是在单独使用时，相互不承担任何责任的协议也是可能的。在这种情况下，每一个版权所有者都可以单独使用作品而无需征得其他版权人的同意。这就好像每一个人都单独拥有版权一样，他们之间也不需要共享收益或相互取得许可。如果您和其他人一起为作品集创作一个游戏而不是为了获益的话，这种合约将是一个很有用的安排。
- 版权所有者具有独有的使用、发布和修改作品的权限。如果您创作的作品集成了其他人作品的一部分或全部，那么就说明这是一个衍生作品。您必须具有原作品版权所有者的使用许可才能合法地出版衍生作品。
- 版权法在作品的合理使用(Fair Use)方面有一个例外。就是在某些情况下，您可以不需要得到版权所有者的许可就能使用某一个作品。这只限制在有限的几种使用上，例如模仿、教学、重要的评论和新闻。在合理使用教条中也有一些需要小心预防的地雷。一般来说，在游戏中集成他人游戏素材的时候，不应该依赖于合理使用教条。
- 由政府创作的作品是没有资格获得版权的。这也是为什么由 NASA 拍摄的太空图片会流传在公共领域(Public Domain)的原因。如果您正在创作的是一个需要恒星、行星、航天飞机、火箭和宇航员图片的科幻游戏，那么只能说您很幸运。登录 <http://gimpsavvy.com/PHOTO-ARCHIVE/>，就可以获取公共领域中太空照片的一个全集，并且在您的程序中可以随意使用。
- 许可(license)是版权所有者对允许使用作品的一个授权。许可有各种形式。有一些许可是独占式的，这意味着只有获得许可的那一个人才能使用这个作品；而有些许可是非独占式的，这意味着很多人可以同时使用某个作品。如果您将游戏的许可独占式地授予了某个人一个有限的时间，那么就相当于您就保留了版权，但是在这段时间内，您也不能将使用该游戏的许可授予其他人。
- 常有的错误理解是：如果有些资料在网络上，而这些资料上又没有版权声明，那么它们就属于公共领域，就可以无需获得许可和归属权直接使用。这是一个低级的错误。一般而言，如果您没有从版权所有者那里明确地得到声明的话，都不能使用这个作品。

这方面的书籍中，我最喜欢的是由 Stephen Fishman (Nolo Press, 2002)编写的 *Web & Software Development: A Legal Guide*。Nolo 出版社出版了很多非常有价值的、自助式的法律图书。我建议在您职业生涯的早期，至少应该阅读一本关于版权方面的书籍。这对于理解您老板的协议、客户的合同和集成开放源代码软件而言都是非常重要的。

1.7.2 选择一个许可

就开放源代码许可的内涵来说，示例游戏代码对您而言相当于是给您授予了许可的。开放源代码许可授予您免费使用和修改源代码、创建自己游戏的权限。如果有些内容不同，通常也是在您起诉开放源代码的创作者(由于开放源代码给您带来的破坏)的能力上设置一些限制。现在，Internet 基础结构的绝大部分代码都依照开放源代码许可进行发布。

虽然不正式地称为商标，但是开放源代码通常专指按照称为 OSI(Open Source Initiative)的组织批准的某一个许可进行发布的代码。在 OSI 网站¹上，可以找到这些经过批准的许可的一个列表，还有开放源代码背后理论方面的一些辅助信息。

¹ <http://www.opensource.org/>

有些开放源代码许可被认为是“病毒携带者”，因为如果您在程序中按照条款集成了其中任何已被许可的代码，它们就被认为是“感染”了您的程序。在这种情况下，您也必须完全发布整个游戏的代码——更大的作品——依照相同的许可条款。这种观念是，没有人可以从这种免费软件中获益，除非他们情愿共享他们对这种软件的修正和贡献。另一方面，开发人员在程序也可以使用非病毒式的许可，在这种程序中，代码的不同部分是使用不同的证书条款进行发布的，包括最终的商业源代码。

在病毒性和非病毒性开放源代码许可的条款之下，可以获得可重用的游戏代码库和示例游戏。请在发布的 `license` 子目录中寻找能使用的不同许可。具体选择哪一个许可要取决于对需求的判断，但是如果必须使用这些代码的话，就必须选择一个许可。

如果还是犹豫不决，我推荐您使用 Lawrence E. Rosen 的 AFL 2.0(Academic Free License)。它的限制可能是最少的。例如，它是非病毒性的，因此，您可以选择将其集成到游戏的最终源代码商业版本中去。其他可能的选择还有 OSL(Open Software License)、GPL(GNU General Public License)和 LGPL(GNU Lesser GPL)。

```
My Cool Game v1.0
Copyright 2004 John Doe.
```

```
Portions Copyright 2003 CroftSoft Inc.
Licensed under the Academic Free License version 2.0
http://www.croftsoft.com/
```

注意，大部分开放源代码许可都需要您为所有包含的或修改的代码维护其归属和版权通告。只要在游戏启动的时候，能在屏幕背景上使用 `system.out` 打印出类似于前面这样的一些内容，对我来说就足够了。

1.7.3 重命名修改后的代码

如果修改了游戏库中的某一个文件，我建议您更改这个包的名称，这样更改过的类就不在 `com.croftsoft` 包的层次结构中了。一般的规则是，您应该将惟一的域名称颠倒过来作为包前缀的开头，以避免名称冲突。例如，我的域名称是 `croftsoft.com`，因此具有惟一性的包名称前缀就是 `com.croftsoft`。

如果没有可以使用的域名，应该考虑获取一个自己的域名。现在从像 Go Daddy Software 这样的注册网²上注册域名称的花费，大约是每年 8.95 美元。至于为您的 Web 页面和 applet(应用小程序)申请网络主机空间的花费，有些提供者的收费是每月不到 8 美元。如果您只想要一个域名称，而不希望花费网络托管费用的话，可以从您的名称注册机构那里使用域转发服务，将访问者重定向到您的个人网页上，该个人网页是您的 Internet 服务提供者(Internet Service Provider, ISP)账户免费自带的。

我还建议(但是不是要求)您应该在修改后的源代码文件中，将我的名字作为其中的一个作者，通过 javadoc 注释 `@author` 的形式保留下来。每当有新的投稿者修改这个文件的时候，`@author` 标记的列表就应该相应地增加。

2 <http://www.godaddy.com/>

1.7.4 共享源代码

除了 CroftSoft 代码库以外，从开放源代码仓库中，还可以找到适合您游戏开发需要的源代码。例如，开放源代码仓库 SourceForge.net 上就有很多不同 Java 游戏的编程项目³。除了进行了较好的分类以外，它仍以铸造场(foundry)的形式对代码进行了组织，例如游戏铸造场(Gaming Foundry)和 Java 铸造场(Java Foundry)。请参阅含有这两部分的项目。

FreshMeat.net 上有一个令人记忆深刻的查询过滤功能，例如，这个功能允许检索分别使用 OSI 批准的开放源代码许可、使用 Java 编程语言和在描述中含有“游戏(game)”单词的项目。这种检索也包含了其他网站(例如 SourceForge.net)上的项目。

最近，Sun Microsystems 已经启动了 Java Games and Game Technologies Projects 网站，这个网站现在正在吸收人们提供的一些开放源代码⁴。我忠心地希望这个网站将来能够成为这种类型代码的中心，作为 Java 游戏铸造厂(Java Gaming Foundry)提供服务——如果它还在 SourceForge.net 网站上的话。但是 SourceForge.net 可能继续占有统治地位，因为它对能够创建什么样的项目和怎样组织这些项目的限制更少。

游戏开发人员 Java 用户组(Game Developers Java Users Group, GameJUG)有一个电子邮件列表(gamejug-open)，这个邮件列表专门用于开放源代码的 Java 游戏开发工作的讨论⁵。我建议您应该订阅一个或多个 GameJUG 邮件列表，例如 gamejug-announce 和 gamejug-news。作为 GameJUG 的志愿者，我和其他同行都经常发表一些与 Java 游戏编程行业相关的、很有价值的信息和链接，这些信息在其他地方您可能根本就找不到。

1.8 为游戏获取多媒体资源

游戏开发小组一般都是由程序员、画面设计师、音频工程师和游戏设计师组成的。如果您一个人创作游戏，那么就必須担当全部的这些角色。这意味着您需要拿出您自己的图片和音频文件。即使您是和其他人协同工作的，您可能也想用一个临时的图像或音频文件进行占位。

1.8.1 图片资源

如果没有专门去请个画面设计师，而自己也没有能力和时间去创作图片，这里有两个方法可以供您选择。第一个选择是使用免费的图片(如图 1-2)。在示例游戏中使用的所有图片文件，您都可以从本书的网站上进行下载使用。这些图片已经提供给了公共领域，这意味着您可以在您自己的游戏中随意地使用它们而不需要付任何版税，甚至不需要说明您是从哪里获得这些图片的。如果您对这些图片进行了足够地修改，那么您对这种衍生的作品就拥有了版权。

Ari Feldman 是 *Designing Arcade Computer Game Graphics* 这本书的作者，他在他的网站⁶上提供了免费的精灵图片库 SpriteLib GPL。请您参阅 zip 文档中的 License.txt 文件，以获取使用条款。注意这里只取用首字母的缩写 GPL，它在这里指的不是 GNU General Public License。

3 <http://www.sourceforge.net/>

4 <http://games.dev.java.net/>

5 <http://www.GameJUG.org/>

6 <http://www.arifeldman.com/>

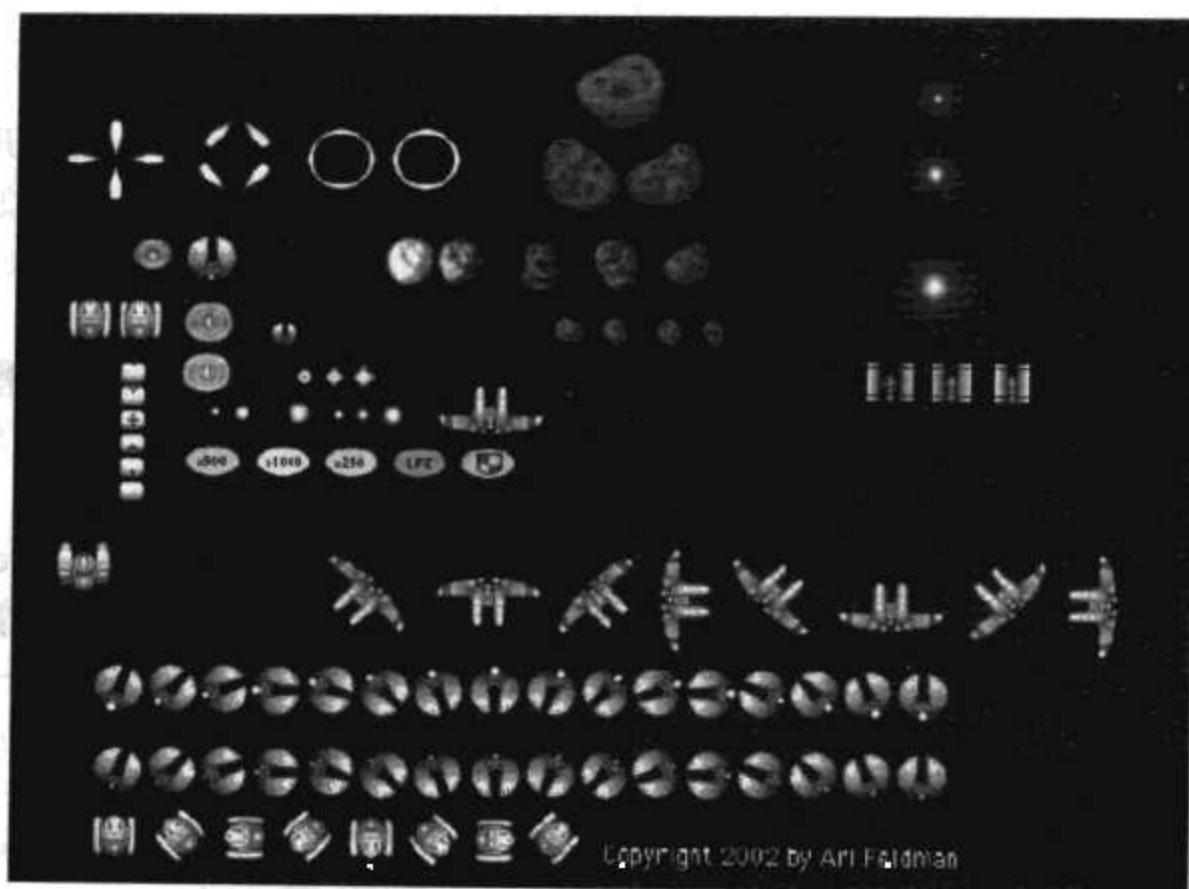


图 1-2 由 Ari Feldman 创作的精灵图片

游戏开发公司 Dimensionality 的董事长和他的长期搭档 James Gholston, 已经在 Public Domain 上发布了他的一些专业图片文件, 因此, 这些图片文件可能也会包含在本书的游戏示例游戏中。从 Dimensionality 的网站⁷或本书的网站上, 都可以获取这些图片。

另外的一个免版权图片的资源是 Clip Art 收集库。根据代码许可条款中的重用这一条, 与开发源代码游戏一起发布的所有图片也是可用的。当在游戏中集成这些免费图片的时候, 您一定要确保自己清楚了使用条款或许可协议, 因为很多资源的免费使用仅限于有限的个人使用或非商业性的使用。

第二个选择就是使用数码相机。这些图像可能不是虚幻的, 但是同时它也很难变得更加真实。简单地拍一张想在游戏中使用的照片, 再按照一定的比例进行缩放和剪裁, 然后再在背景上做一些透明化的处理。为了能够进行这些处理, 可能需要使用的开放源代码的图像编辑工具就是 GIMP(GNU Image Manipulation Program)。GIMP 可以在 Unix 和 Windows 这两种操作系统上使用⁸。

如果确实想作一些自己的艺术创作, 也可以使用 GIMP 创作一些新的图片。我经常使用 Windows 自带的 MS Paint 附件, 创作简单的占位图片。我注意到了与 Windows XP 一起发布的很多最新版本的画图工具, 现在已经可以保存 PNG 格式的图片。我们还可以获取其他一些免费工具, 例如 Project Dogwaffle 和 Pixia⁹。

1.8.2 音频资源

本书示例游戏中使用的所有音频文件, 都可以从本书的网站下载使用。这些音频资源也已经提供给了公共领域, 您可以使用或修改它们而不用承担任何责任和义务。

⁷ <http://www.dimensionality.com/free.html>

⁸ <http://www.gimp.org/>

⁹ <http://www.squirrelldome.com/cyberop.htm>, <http://www.ab.wakwak.com/~knight/>

Cosmi 中的音效库 6000 Sound Effects 是程序可以使用的另外一个资料资源¹⁰。这个包在 Amazon.com 上的售价是 9.99 美元。在销售网站上已经声明了,您可以在您的商业游戏中使用这些音效文件而不用交付任何版税。记住,如果将这些文件和开放源代码游戏一起发布的话,可能会使许可复杂化。

如果您想录制自己的声音,所需要的就是一个麦克风。在音频编辑器的使用方面,我没有太多的经验,但是我曾成功地使用过 GoldWave 的共享版¹¹。您可能会经常想使用这样的一个编辑器来将您录制的音频文件剪辑为适合您游戏长度的一些样本。

1.9 基本示例

假设已经建立了开发环境,那么在学习了第1章以后,就应该能够开始制作您自己的游戏了。可以修改如图1-3所示的示例游戏来开始您的创作。

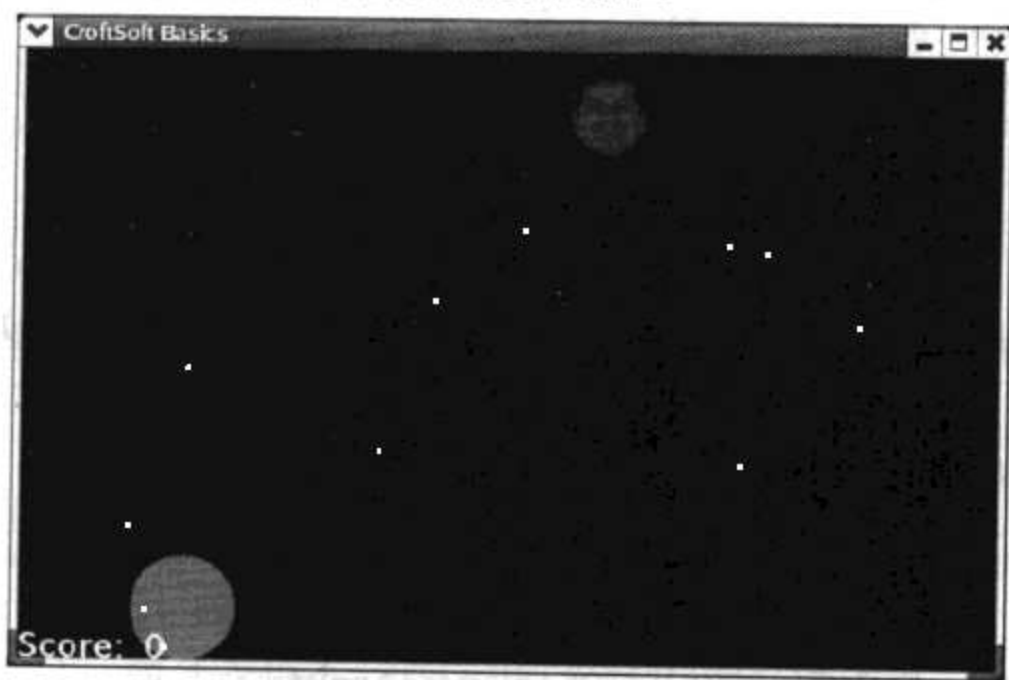


图 1-3 基本示例

这个简单游戏就是将小球向屏幕顶部的一个移动目标发射。可以使用 Ant 的目标 basics 来构建和启动这个示例游戏。它使用了极少量的代码,介绍了加载媒体文件、绘制图形和图像、显示文本、播放声音、产生随机数、维持状态、动画、鼠标和键盘输入、冲突检测和脏矩形管理等实现方法。

1.9.1 修改源代码

这个游戏是在一个扩展游戏库的基础上构建的,在后续的章节中我还要对这个扩展游戏库进行介绍。在回顾这些源代码的时候,我会指出哪些代码您现在就可以安全地进行修改,哪些代码是需要您进行进一步学习以后才能进行修改的。由于这个示例几乎涵盖了游戏编程的全部基础知识,很多读者可能会发现他们能通过简单地复制和修改这个方法,不需要学习第2章,就可以成功地创建游戏。

```
package com.croftsoft.ajgp.basics;
```

¹⁰ <http://www.cosmi.com/>

¹¹ <http://www.goldwave.com/>

可以在 `com.croftsoft.ajgp.basics` 包中找到这个游戏。需要修改的第一项内容就是包的名称。

```
import java.applet.Applet;
import java.applet.AudioClip;
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.util.Random;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JComponent;
```

先删除这些import语句。注意，只导入了3个自定义的类。其余的都是从Java内核中导入的。

```
public final class BasicsExample
    extends AnimatedApplet
{
    //////////////////////////////////////
    //////////////////////////////////////
}
```

更改这个类名称，但是要继续对其超类AnimatedApplet进行扩展。

```
private static final String VERSION
    = "2003-11-06";

private static final String TITLE
    = "CroftSoft Basics";

private static final String APPLET_INFO
    = "\n" + TITLE + "\n"
    + "Version " + VERSION + "\n"
    + CroftSoftConstants.COPYRIGHT + "\n"
    + CroftSoftConstants.DEFAULT_LICENSE + "\n"
    + CroftSoftConstants.HOME_PAGE + "\n";
```

更改VERSION、TITLE和APPLET_INFO的值。

```
private static final Color      BACKGROUND_COLOR
    = Color.BLACK;

private static final Cursor     CURSOR
    = new Cursor ( Cursor.CROSSHAIR_CURSOR );
```



```

private static final Font      FONT
    = new Font ( "Arioso", Font.BOLD, 20 );

private static final Double    FRAME_RATE
    = new Double ( 30.0 );

private static final Dimension FRAME_SIZE
    = new Dimension ( 600, 400 );

private static final String    SHUTDOWN_CONFIRMATION_PROMPT
    = "Close " + TITLE + "?";

```

可以指定背景色、鼠标指针的类型、文本字体、动画速度、窗口大小和关闭确认提示。这些常量都被传递给它的超类 `AnimatedApplet`。您或许想保持这些常量，并按照需要定制这些常量的值。

```

private static final String MEDIA_DIR = "media/basics/";

private static final String AUDIO_FILENAME
    = MEDIA_DIR + "drip.wav";

private static final String IMAGE_FILENAME
    = MEDIA_DIR + "croftsoft.png";

```

可以根据需要添加或删除这些媒体文件的名称。

```

private static final long    RANDOM_SEED = 0L;

private static final Color   BALL_COLOR = Color.RED;

private static final Color   SCORE_COLOR = Color.GREEN;

private static final int     VELOCITY = 3;

```

还可以根据游戏的需要添加或删除附加常量。可以与随机数生成器一起使用 `RANDOM_SEED`。`VELOCITY`是保龄球的移动速度，单位是每帧多少像素。

```

private final Rectangle componentBounds;

private final Random    random;

```

变量 `componentBounds` 是存储游戏屏幕的宽和高的一個 `Rectangle`。变量 `random` 是一个随机数生成器。在很多游戏中，可能都需要这些变量。

```

private final Rectangle ballRectangle;

private final Rectangle targetRectangle;

```

变量 `ballRectangle` 和 `targetRectangle` 是用于冲突检测的。这些变量是游戏特定的。

```

private boolean    componentResized;

```

```
private KeyEvent  keyEvent;

private Point     mousePoint;

private boolean   mousePressed;
```

这些变量用于标记或存储用户的输入事件。在大多数程序中，都可能需要使用它们。

```
private AudioClip audioClip;

private Icon       icon;
```

对使用的每一个媒体文件，都会有一个audioClip或icon。

```
private boolean   rolling;

private int        score;
```

这些是游戏特定的状态变量。变量rolling指示小球是否已经发射。

```
public static void main ( String [ ] args )
///////////////////////////////////////////////////////////////////
{
    launch ( new BasicsExample ( ) );
}
```

在main()方法中，将BasicsExample更改为新类的名称。

```
private static AnimationInit createAnimationInit ( )
///////////////////////////////////////////////////////////////////
{
    AnimationInit animationInit = new AnimationInit ( );

    animationInit.setAppletInfo ( APPLET_INFO );

    animationInit.setCursor ( CURSOR );

    animationInit.setFont ( FONT );

    animationInit.setFrameIconFilename ( IMAGE_FILENAME );

    animationInit.setFrameRate ( FRAME_RATE );

    animationInit.setFrameSize ( FRAME_SIZE );

    animationInit.setFrameTitle ( TITLE );

    animationInit.setShutdownConfirmationPrompt (
        SHUTDOWN_CONFIRMATION_PROMPT );

    return animationInit;
}
```


静态方法 `createAnimationInit()` 设置了许多由超类 `AnimatedApplet` 常用的参数。可能不需要修改这个方法。

```
public BasicsExample ( )
///////////////////////////////////////////////////
{
    super ( createAnimationInit ( ) );

    componentBounds = new Rectangle ( );

    random = new Random ( RANDOM_SEED );
```

将构造函数的名称更改为新名称。记住，对超类构造函数的首次调用也是如此。您几乎是一直都需要这个 `componentBounds` 变量。很多游戏都使用随机数生成器。

```
animatedComponent.addComponentListener (
    new ComponentAdapter ( )
    {
        public void componentResized (
            ComponentEvent componentEvent )
        {
            componentResized = true;
        }
    } );

animatedComponent.addKeyListener (
    new KeyAdapter ( )
    {
        public void keyPressed ( KeyEvent ke )
        {
            keyEvent = ke;
        }
    } );

animatedComponent.addMouseListener (
    new MouseAdapter ( )
    {
        public void mousePressed ( MouseEvent mouseEvent )
        {
            mousePressed = true;
        }
    } );

animatedComponent.addMouseMotionListener (
    new MouseMotionAdapter ( )
    {
        public void mouseMoved ( MouseEvent mouseEvent )
        {
            mousePoint = mouseEvent.getPoint ( );
        }
    } );
```

这些方法将用户输入事件的侦听器附加给 `animatedComponent` 变量。变量 `animatedComponent` 是从其超类 `AnimatedApplet` 继承过来的。一般情况下，您会希望拥有这些事件侦听器。

```
ballRectangle = new Rectangle ( );

targetRectangle = new Rectangle ( );
}
```

可以在构造函数中初始化游戏特定的多个对象。

```
public void init ( )
///////////////////////////////////////////////////
{
    super.init ( );

    animatedComponent.requestFocus ( );

    componentResized = true;
```

像在第2章中介绍的一样，在首次启动游戏的时候，就会调用 `init()` 方法，执行没有在构造函数中执行的一些其他初始化过程。通常我们总会希望调用超类的 `init()` 方法，请求键盘焦点，以及将 `componentResized` 设置为 `true`。第一次将屏幕的宽度和高度加载到 `componentBounds` 变量中时，都必须设置 `componentResized` 标志。

```
ClassLoader classLoader = getClass ( ).getClassLoader ( );

audioClip = Applet.newAudioClip (
    classLoader.getResource ( AUDIO_FILENAME ) );

icon = new ImageIcon (
    classLoader.getResource ( IMAGE_FILENAME ) );
```

由于在后面章节中将要解释的一些原因，在调用初始化方法 `init()` 以后，才会创建一些媒体对象，否则，就会抛出 `NullPointerException` 异常。这个示例演示了如何将音频文件和图像文件数据加载到内存中。

```
ballRectangle.width = icon.getIconWidth ( );

ballRectangle.height = icon.getIconHeight ( );

targetRectangle.width = icon.getIconWidth ( );

targetRectangle.height = icon.getIconHeight ( );
}
```

小球和目标的大小要与图像的大小相等。

```
public void update ( JComponent component )
```



```

////////////////////////////////////
{

```

正如将要在第3章介绍的一样，`update()`方法是执行全部状态更新的方法。每当游戏循环请求动画的一个新帧时，都会调用这个方法。游戏循环逻辑是由超类`AnimatedApplet`提供的。

```

if ( componentResized )
{
    componentResized = false;

    component.repaint ( );

    component.getBounds ( componentBounds );

    if ( !rolling )
    {
        ballRectangle.y = componentBounds.height - ballRectangle.height;
    }
}

```

当游戏玩家调整运行这个游戏的桌面窗口大小的时候，就会产生一个事件。这个事件会被在构造函数中创建的一个事件侦听器截取。对用户输入事件的处理应该一直被延迟，直到调用了 `update()` 方法。由于这个原因，在构造函数中创建的所有事件侦听器都只是简单地标记或存储事件，而不是立即去处理这些事件。

其中的一个事件侦听器会增加布尔标志 `componentResized` 的值，表明游戏屏幕的大小已经发生了改变。您总是希望在您的 `update()` 方法的实现中，通过重新将标志的值设为 `false`、请求重画整个屏幕以及像前面代码那样在 `componentBounds` 中存储新窗口的大小的方法，来处理这个事件。

然后，您也许希望提供附加的游戏特定逻辑来调整基于新窗口大小的状态坐标。在这个示例中，如果小球还没有发射的话，它的位置就是在屏幕的底部。

```

boolean rollRequested = false;

if ( mousePressed )
{
    mousePressed = false;

    rollRequested = true;
}

```

当用户按下鼠标的时候，就会产生一个事件。这个事件被在构造函数中创建的一个鼠标事件侦听器截获。鼠标事件侦听器会增加布尔标志 `mousePressed` 的值，这样，在稍后调用 `update()` 方法时，这个事件就会得到处理。在这个示例中，当按下鼠标按钮的时候，变量 `rollRequested` 的值就会设置为 `true`，表示小球应该向着目标进行发射。

```

int ballMove = 0;

if ( keyEvent != null )
{

```

```

int keyCode = keyEvent.getKeyCode ( );
switch ( keyCode )
{
    case KeyEvent.VK_SPACE:

        rollRequested = true;

        break;

    case KeyEvent.VK_LEFT:
    case KeyEvent.VK_KP_LEFT:

        ballMove = -1;

        break;

    case KeyEvent.VK_RIGHT:
    case KeyEvent.VK_KP_RIGHT:

        ballMove = 1;

        break;
}

keyEvent = null;

mousePoint = null;
}

```

也可以通过按下键盘上的空格键来发射这个小球。按左右箭头分别将 ballMove 变量设置为 -1 和+1。在 update()方法处理键盘事件以后，就会将 keyEvent 的值设置为 null，这样在游戏循环再次调用 update()方法之前，就不会处理相同的键盘事件。只要产生了一个 keyEvent 事件，鼠标事件 mousePoint 就会被设为 null，因为键盘的优先级比鼠标要高。

```

if ( mousePoint != null )
{
    if ( mousePoint.x
        < ballRectangle.x + ballRectangle.width / 2 - VELOCITY)
    {
        ballMove = -1;
    }
    else if ( mousePoint.x
        > ballRectangle.x + ballRectangle.width / 2 + VELOCITY )
    {
        ballMove = 1;
    }
    else
    {
        mousePoint = null;
    }
}

```


在将小球发射出去以前，还可以使用鼠标向左或向右移动它的位置。当玩家移动鼠标的时候，在构造函数中创建的一个事件侦听器就会以 `mousePoint` 的形式存储鼠标指针的位置。每次调用 `update()` 方法时，就会将球心向最新产生的 `mousePoint.x` 值的位置靠近。一旦球心到达 `mousePoint.x`，或接近只要再继续移动就能命中目标的时候，`mousePoint` 的值就被置为 `null`。

```
if ( rollRequested )
{
    if ( !rolling )
    {
        audioClip.play ( );

        rolling = true;
    }
}
```

如果玩家已经通过单击鼠标或按下键盘上的空格键，让小球向目标方向进行发射，那么如果小球这时还没有开始滚动(`rolling`)，程序就会播放一个声音文件。

```
component.repaint ( targetRectangle );
```

正如在后面章节中介绍的一样，`update()` 方法必须请求对象在它们的原位置和新位置上进行重绘，以产生动画效果。在这个示例中，重绘的区域包含被请求目标的原位置。每一次调用 `update()` 方法的时候，都会做这种重绘工作，因为目标一直在移动。

```
if ( rolling )
{
    component.repaint ( ballRectangle );

    ballRectangle.y -= VELOCITY;
```

如果小球是朝着目标的方向滚动的，那么包含它原位置的屏幕区域也会要求进行重绘，它的 `y` 坐标是通过它的 `VELOCITY` 进行递减的。

```
boolean reset = false;

if ( targetRectangle.intersects ( ballRectangle ) )
{
    reset = true;

    targetRectangle.x = -targetRectangle.width;

    audioClip.play ( );

    score++;

    component.repaint ( );
}
```

如果滚动中的小球与目标发生了碰撞，游戏就被重新设置，同时还会播放一个声音文件，得分也会进行递增，并且要求重画整个屏幕。

```

else if ( ballRectangle.y + ballRectangle.height < 0 )
{
    reset = true;

    if ( score > 0 )
    {
        score--;
    }

    component.repaint ( );
}

```

如果小球还在滚动，但是已经错过了目标，游戏就会被重置，但是得分会递减。这时也会要求重画整个屏幕，更新后的得分也会显示出来。

```

if ( reset )
{
    ballRectangle.y = componentBounds.height - ballRectangle.height;

    rolling = false;
}

```

如果游戏被重置——无论是由于小球碰到了目标还是错过了目标——小球都会被移动到屏幕底部发射的位置，rolling 状态标志也都会被设为 false。

```

    component.repaint ( ballRectangle );
}

```

由于小球还在滚动，所以在它的新位置上也必须重绘小球。

```

else if ( ballMove != 0 )
{
    component.repaint ( ballRectangle );

    ballRectangle.x += ballMove * VELOCITY;

    if ( ballRectangle.x < 0 )
    {
        ballRectangle.x = 0;
    }

    if ( ballRectangle.x
        > componentBounds.width - ballRectangle.width )
    {
        ballRectangle.x
            = componentBounds.width - ballRectangle.width;
    }

    component.repaint ( ballRectangle );
}

```


如果小球并没有向目标方向滚动，那么它可能在发射区域内按照玩家瞄准的方向向左或向右移动。这是由 `ballMove` 的一个非零的值指示的。小球不能移动到屏幕左边或右边的边界以外的地方去。在小球的原位置和新位置上会请求对小球进行重绘。

```
if ( score > 1 )
{
    targetRectangle.x += random.nextInt ( score ) * VELOCITY;
}
else
{
    targetRectangle.x += VELOCITY;
}
```

随着玩家得分的增加，目标的水平速度也变得更加具有随机性。这使得游戏对玩家而言更加具有挑战性。

```
if ( targetRectangle.x >= componentBounds.width )
{
    targetRectangle.x = -targetRectangle.width;
}
```

如果目标移动到屏幕右边边界以外，它就会接着再从左边的边界进入屏幕，这样目标就会连续地从左到右在屏幕的顶部穿行。

```
component.repaint ( targetRectangle );
}
```

`update()`方法的最后一个动作是要求重绘包含目标新位置的屏幕区域。在`update()`方法中，先进行原位置的重绘请求。

```
public void paint (
    JComponent component,
    Graphics2D graphics )
{
    //////////////////////////////////////
```

如将在第3章中介绍的一样，只要屏幕需要进行重绘，就会调用 `paint()`方法，而且通常在动画过程中，每个游戏循环调用一次。`paint()`方法的调用是由超类 `AnimatedApplet` 进行管理、由 `update()`方法中作出的重绘请求进行触发的。

```
graphics.setColor ( BACKGROUND_COLOR );
```

```
graphics.fill ( componentBounds );
```

总是首先绘制出背景。

```
icon.paintIcon ( component, graphics, targetRectangle.x, 0 );
```

再绘制目标。

```
graphics.setColor ( BALL_COLOR );
```

```
graphics.fillOval (
    ballRectangle.x,
    ballRectangle.y,
    ballRectangle.width,
    ballRectangle.height );
```

再绘制小球。

```
graphics.setColor ( SCORE_COLOR );

graphics.drawString (
    "Score: " + Integer.toString ( score ),
    0, componentBounds.height );
}
```

最后，显示出得分文本。因为在 `paint()` 方法中，得分文本是在小球之后才显示出来的，所以得分会被小球阻挡住。

注意，在 `paint()` 方法中，并没有更新游戏的状态，在 `update()` 方法中，也没有要绘制的图形。这也是另一个要遵守的约定。

1.9.2 修改构建文件

在测试完游戏示例和对其源代码进行了回顾以后，您可能想继续尝试，并将这个游戏作为您自己开发游戏的模板。我建议您采用一个逐步扩充的方法来做这种开发，在进行了每一步较小的更改以后，都要进行构建和代码测试。您首先想做的更改可能就是为类更改包的名称。虽然这只是对一行代码进行的更改，但是，如果您没有考虑到需要对 Ant 构建文件进行同步更改的话，一行代码的更改也会使您遭遇失败。下面是需要进行更改的代码文档。

```
<target name="basics_prep" depends="init">
  <copy todir="${tmp_dir}/media/basics">
    <fileset dir="${res_dir}/ajgp/basics/media">
      <include name="croftsoft.png"/>
      <include name="drip.wav"/>
    </fileset>
  </copy>
</target>
```

如果从游戏中添加或删除了媒体文件，就必须修改目标 `basics_prep`。

```
<target name="basics" depends="basics_prep">
  <javac srcdir="${src_dir}" destdir="${tmp_dir}">
    <include name="com/croftsoft/ajgp/basics/BasicExample.java"/>
  </javac>
  <echo
    file="manifest.txt"
    message="Main-Class: com.croftsoft.ajgp.basics.BasicExample" />
  <jar
    basedir="${tmp_dir}"
    destfile="${arc_dir}/basics.jar"
    manifest="manifest.txt"
```



```

        update="false"/>
        <delete file="manifest.txt"/>
        <delete dir="${tmp_dir}"/>
        <java fork="true" jar="${arc_dir}/basics.jar"/>
    </target>

```

如果更改了包或类的名称，就必须更改目标 `basics`。

到此为止，我们已将 Java 游戏编程的 `basics` 示例介绍完毕。如果其中还有些内容让您觉得不明白也没有关系，因为在下面的章节中我还要对其进行更加详细的介绍。后续章节在将这段代码作为一个有用的示例和原始模板的同时，对其中的代码作了很多的改进和提高。

1.10 小结

本章主要介绍了怎样使用开发构建工具 `Ant`，编译本书所使用的示例代码。还介绍了其他一些开发工具，例如图像和音频编辑器。另外，还指出了一些代码和多媒体资源，您可以将这些资源集成到您的游戏中，其间还附加介绍了开放源代码的许可条款问题。本章最后还给出了一个示例游戏的源代码，用来介绍 Java 游戏程序编写的基本知识，这个示例游戏还可以作为一个游戏模板进行重新利用。在后续的章节中，将详细介绍可重用游戏库中可用的一些类。

1.11 参考文献

Eckstein, Robert. *XML Pocket Reference, 2nd edition*. Sebastopol, CA: O'Reilly & Associates, 2001.

Feldman, Ari. *Designing Arcade Computer Game Graphics*. Plano, TX: Wordware Publishing, 2000.

Fishman, Stephen. *The Public Domain: How to Find & Use Copyright-Free Writings, Music, Art & More*. Berkeley, CA: Nolo Press, 2001.

Fishman, Stephen. *Web & Software Development: A Legal Guide, 3rd edition*. Berkeley, CA: Nolo Press, 2002.



第 2 章 部署 框架

勤奋是好运之母。

——本杰明·富兰克林

本章将介绍如何使用不同的部署框架，采用多种不同的方法部署 Java 游戏。这里所谓的框架(framework)，实际上指的是容器在不同平台上部署、启动、运行和优雅终止游戏的标准化机制。框架包括浏览器的 applet、可执行的 JAR 文件和 Java Web Start 这 3 种。这里将详细地介绍一个可重用框架，这个可重用框架使我们无须重新编译，就可以将游戏部署到这 3 种环境中(实际上也只有这 3 种环境)。本章是通过提供进行动画线程管理所需要的一些方法，并在一个通用的接口中定义这些方法来实现这种可重用框架的。

2.1 部署为 applet

通常，人们认为 applet 就是用于显示 Web 页面中嵌入式动画的 Java 小程序。这种理解实际上并不准确，因为可以对除动画之外的其他大型程序中使用 applet，并且也可以在容器中使用它，而不是仅限于在 Web 浏览器中使用。但是，对所有 applet 来说，相同的一点就是它们都扩展了 java.applet.Applet 类，这个类使我们可以所有兼容 applet 的框架中使用它们。

2.1.1 实现生命周期方法

几年以前，在设计模式(Design Pattern)课程中，我曾学过框架这个术语的定义，这个定义我永远都不会忘记。教授简单地这样介绍，“它是一个好莱坞模式——Don't call us, we'll call you”。对一个要初始化、启动、停止和销毁游戏对象的容器而言，它必须有访问那些能按需在游戏对象上调用预定义方法的那种权限。因为在一个给定的框架内，这些生命周期方法都是一致的，所以针对某个框架编写的所有游戏程序，都能在与框架兼容的容器中运行，而与供应商和平台无关。

java.applet.Applet 类定义了 4 个生命周期方法：init()、start()、stop()和 destroy()，这些生命周期方法都是由与框架兼容的容器进行(例如 Web 浏览器)调用的。Applet 的子类通过重载其中的一个或多个空方法来创建动画。语义方面有些微妙，所以我建议您应该回顾一下这些方法的 Javadoc。下面是一些补充说明。

1. init()方法

init()方法用来初始化游戏。在这个方法中，可以加载图片、音频文件以及其他的一些静态信息。可以在这个方法中创建在游戏启动以前需要产生的所有动态数据。还可以在这个方法中，加载以前保存的一些历史数据，例如游戏玩家的最高得分等。

记住，在同一个 applet 实例中，永远都不会两次调用 `init()` 方法。在 Applet 子类中，在 `init()` 方法之前不会调用其他方法(除了不带任何参数的构造函数以外)。这说明我们不必担心容器会在 applet 准备好之前就去调用它的 `paint()` 方法来显示它。

以前总是有人不断地提醒我：必须在 `init()` 方法中执行所有初始化的代码，而不能在构造函数中去做这些初始化的工作。实际上，在 Java 1.4 之前，Applet 类甚至还没有定义一个可以在子类的实现中重写的公共构造函数。

2. `start()` 方法

`start()` 方法启动驱动动画的游戏循环线程。这个方法不仅可以用于启动动画，或其他 applet 正在执行的那些事情，它甚至还可以在动画暂停以后重新启动动画。容器可能会重新启动 applet 处理的一个典型示例，就是当玩家将页面窗口进行最大化或最小化的时候。在每一个 applet 实例的生命周期内，包容器都可能会多次调用 `start()` 方法。

实现 Applet 的子类时，在容器首次调用 `start()` 方法以前，应该准备接收一个容器(通常情况下，这个容器就是浏览器)对 `paint()` 方法的调用。在动画线程启动以前，代码可能会被调用去显示动画的第一个帧画面，或其他某些需要显示的内容，这种想法并不是一种直觉。如果在首次加载 applet 的时候，而不是在已经运行了一段时间以后，代码偶尔或甚至是经常性地从 `paint()` 方法中引发 `NullPointerException` 异常，那么可以估计在 `start()` 方法中有一些对图片进行初始化的代码。将这些代码转移到 `init()` 方法以后，问题会马上消失。

3. `stop()` 方法

在游戏暂停时，需要使用 `stop()` 方法来挂起动画。根据方法定义的不同，在 applet 实例的整个生命周期内，容器可能会多次调用 `stop()` 方法。容器能够多次调用 `stop()` 而不需要介入调用 `start()` 吗？容器能够多次调用 `start()` 而不需要介入调用 `stop()` 吗？这个问题的答案我也不太确定，因此我通常是采用一种不会带来什么麻烦的方式，编写这两种方法的实现。

还需要注意的是，在调用 `destroy()` 方法以前，必须保证正确调用了 `stop()` 方法。我习惯在两种假定下——容器可能已经调用了 `stop()` 方法，或可能还没有调用 `stop()` 方法——实现 `destroy()` 方法。之所以这么做是因为我不相信容器能够正确地实现了框架约定，尤其是如果我自己创建了一个可能含有一些 bug 的 applet 容器时。这样做还因为调用 `stop()` 的时候也可能失败和引发异常。

可以编写一个不正确的 `stop()` 方法。例如，当由容器调用该 `stop()` 方法时，它可能会延迟到容器最后杀死这个线程的时候。您的代码也可能会忽略对 `stop()` 方法的调用，而继续执行，将 `stop()` 方法看作一个提示而不是作为一个命令去执行。我不清楚您的情况会怎么样，反正我在实现这种代码的时候，它总是不能实现我的目的。

4. `destroy()` 方法

游戏结束的时候，容器就调用 `destroy()` 方法来取消分配给游戏的所有资源。有一些与 `destroy()` 方法相关的基本事实。在 applet 实例的生命周期内，容器最多只会调用 `destroy()` 方法一次。除了 `finalize()` 方法以外，在 `destroy()` 方法之后就再也不会调用其他什么方法了。容器永远不会通过调用 applet 实例的 `init()` 方法，重新使用已经调用过其 `destroy()` 方法的 applet 实例。

正如 stop()方法一样，您可以编写一个带有 bug 的 destroy()方法的实现，让这种方法不按照您的期望去完成其功能。

您可能还会怀疑为什么需要 destroy()方法。毕竟，当容器丢弃该 applet 的时候，垃圾回收器应该会自动地去回收或关闭所有使用过的资源。这种逻辑带来的问题是垃圾回收器可能不能按照正确的方法处理 applet。这样就说明 applet 可能会继续非确定性地占用一些资源，例如像一些未关闭的 Socket 链接、未清除的 Graphics 上下文和未清除的 Window 本地屏幕资源。如果您注意到 applet 在运行了几次以后就总会出现死掉的情况，但是在您重新启动浏览器以后，这种情况就消失了，这时，就应该怀疑在 applet 生命周期的最后，是否完全释放了初始化的资源。

Object 类的 finalize()方法是留给想要执行一些行为(通常是在对象被垃圾回收器回收以前，所占用资源的释放)的子类去重写的。请不要使用 finalize()方法。这是一个不可靠的方法，因为它不能保证肯定会被调用。如果有足够多的内存，那么垃圾回收器可能就不会去做回收垃圾的工作，因为这样将会是自找麻烦。如果是正在关闭 JVM，那么它也可能不会去调用 finalize()方法。

一般而言，为了及时地释放系统资源，最好的方法就是依靠 destroy()方法。

2.1.2 管理 applet 动画线程

当开始和停止 applet 线程的时候，那些曾要求提供的资源就被禁止了。以前，生命周期方法的实现可能会与下面的过程相似：

```
public void start ()
///////////////////////////////////////////////////
{
    animationThread = new Thread (this);

    animationThread.start ();
}

public void stop ()
///////////////////////////////////////////////////
{
    animationThread.stop ();
}
```

或者与下面代码类似：

```
public void start ()
///////////////////////////////////////////////////
{
    if ( animationThread == null )
    {
        animationThread = new Thread ( this );

        animationThread.start ();
    }
    else
    {
        animationThread.resume ();
    }
}
```



```

    }
}

public void stop ( )
///////////////////////////////////////////////////
{
    animationThread.suspend ( );
}

public void destroy ( )
///////////////////////////////////////////////////
{
    animationThread.stop ( );
}

```

这种技术带来的问题是，使用 `java.lang.Thread` 类的方法 `stop()`、`suspend()` 和 `resume()` 有时会在一个非连续的状态中安放一个对象，或者导致死锁。这可能会产生一些负面效果，使程序难于调试。正是由于这个原因，这些方法都被淘汰了。我已经注意到，在替换了那些使用 `Thread.stop()` 的原有代码以后，一些奇怪的 bug 就消失了。请参考 Sun 的在线文章：[Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated](http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html)¹。

```

public void start ( )
///////////////////////////////////////////////////
{
    stopRequested = false;

    animationThread = new Thread ( this );

    animationThread.start ( );
}

public void stop ( )
///////////////////////////////////////////////////
{
    stopRequested = true;
}

public void run ( )
///////////////////////////////////////////////////
{
    while ( !stopRequested )
    {
        animate ( );
    }
}

```

不用费力地去停止一个线程，完全可以像前面代码一样，使用一个循环轮询布尔停止请求标志，让该线程优雅地死掉。注意，在作这个请求的时候，动画线程并不会立即停止，因为停

¹ <http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html>

止请求标志在每一个循环中只会检查一次。如果在 `stop()` 方法之后但在动画循环有机会轮询这个标志的值以前就马上调用 `start()` 方法，就不会产生什么问题。使用这种技术时，如果您的运气不好，选择的时间又不恰当，就可能终止两个或多个正在同时运行的线程。

```
public void start ( )
///////////////////////////////////////////////////
{
    animationThread = new Thread ( this );

    animationThread.start ( );
}

public void stop ( )
///////////////////////////////////////////////////
{
    animationThread = null;
}

public void run ( )
///////////////////////////////////////////////////
{
    Thread animationThread = Thread.currentThread ( );

    while ( animationThread == this.animationThread )
    {
        animate ( );
    }
}
```

上面代码解决了这个问题。这里，最可贵的方法不是这个共享的布尔标志，而是去判断当前的线程是否就是 `applet` 实例已经将其指定为主动画线程的线程。这个操作是使用身份比较的方法执行的。注意，在 `run()` 方法中，`animationThread` 是一个本地方法引用，`this.animationThread` 是一个对象实例的引用。最坏的情况下，在正常退出以前，对于另一个迭代，原动画线程会与新动画线程重叠。

```
public synchronized void start ( )
///////////////////////////////////////////////////
{
    stopRequested = false;

    if ( animationThread == null )
    {
        animationThread = new Thread ( this );

        animationThread.start ( );
    }
    else
    {
        notify ( );
    }
}
```



```

}

public synchronized void stop ( )
////////////////////////////////////
{
    stopRequested = true;

    animationThread.interrupt ( );
}

public void run ( )
////////////////////////////////////
{
    while ( animationThread != null )
    {
        try
        {
            animate ( );

            if ( stopRequested )
            {
                synchronized ( this )
                {
                    while ( stopRequested )
                    {
                        wait ( );
                    }
                }
            }
        }
        catch ( InterruptedException ex )
        {
        }
    }
}

public synchronized void destroy ( )
////////////////////////////////////
{
    animationThread = null;

    stopRequested = false;

    notify ( );
}

```

这段代码通过使用一个单独的线程——该单独线程使用布尔标志 `stopRequested` 和 `Object` 类的 `wait()` 方法进行挂起和重新启动，演示了怎样防止原动画线程和新动画线程在过渡过程中哪怕是片刻的重叠。注意，这段代码使用了同步功能，这项功能会使整个过程减慢。这应该不是一个主要的问题，因为动画暂停下来以后，在主动画循环内部就不会有同步请求了。

`start()`方法和 `destroy()`方法以及 `run()`方法的一部分都被同步了，因为任何调用 `notify()`方法或 `wait()`方法的线程必须是对象监控器的所有者。`stop()`方法被同步是因为它修改了共享变量 `stopRequested`。

注意，`stop()`方法调用了 `animationThread.interrupt()`。这是因为循环中的 `animate()`方法可能会监控中断的状态，以确定它是否应该提早退出。在长动画帧产生的过程中，它可能会以连续的步骤检查这些状态。如果它在 `animate()`实现内部使用 `Thread.sleep()`进行延时，以将循环的速度降低到与理想的帧速一样的话，它就会检查这些状态。

当 `destroy()`方法不再引用 `animationThread` 的时候，动画循环就会退出。注意，当调用 `destroy()`方法的时候，动画线程不会立即退出，而是在下一个循环的开始处进行检查时退出。这说明循环仍然可能会在下一个循环到最后一个循环的循环体内使用那些资源，在 `destroy()`方法执行完成以后，产生动画。如果容器在 `stop()`方法以后立即调用了 `destroy()`方法，就几乎总会发生这种事情。如果想对前面的 `destroy` 方法添加一些辅助代码来释放动画过程中使用的资源，可能需要将其中的有些代码放置在 `run()`方法的结尾处。

循环内的 `try/catch` 块用于捕获由 `animate()`方法或由 `wait()`方法引发的 `InterruptedExceptions` 异常。注意，由于 `animationThread.interrupt()`调用是在 `stop()`方法内，所以在 `animate()`方法的绝大部分时间内都可能会引发这个异常。线程已经在 `wait` 方法上挂起的时候，一般都不会调用 `stop()`方法。

调用不在其间调用 `start()`方法的 `stop()`方法，使得动画提升了一帧。这是因为 `wait()`方法引发了一个 `InterruptedException` 异常，然后循环就在半个迭代体内继续。在再次在 `wait()`命令上停止以前，它就只调用一次 `animate()`方法。一般地，容器在一个生命周期内不会连续两次调用 `stop()`方法，设计代码的时候，也没有规划这个提升一帧的特征。但是，您仍然可能会发现它很有用。

因为我认为动画线程管理的这最后一个技术是最健壮的，所以选择它用于后面章节中描述的可重用动画库的设计。

2.1.3 读取 JAR 文件

Java 归档文件(JAR)就是一个压缩的 `zip` 文件，这种文件中包含游戏所需要的全部代码和多媒体文件。它大大地减少了启动 `applet` 所需要的总时间，因为玩家可以在单个 `HTTP` 请求中下载游戏。

在 JAR 文件以前，开发人员用于减少 `applet` 下载时间的技术是将所有的图像都复制到一个文件中。如图 2-1 所示，各个小图像就会以网格或带状的形式被编排在较大的组合图中，并安放在不同地方。一旦下载下来，就可使用坐标数据和定制代码从该组合图中不同位置上剪取图像数据的矩形。幸运地是，我们现在有了 JAR 文件，所以就不再需要这种额外的代码了。现在使用它们惟一的名称以及与其归档文件的根目录相关联的目录路径，直接打开它们，就可以很简单地访问每一个图像。如果您有一个以单个图像形式显示的精灵图片，在开发之前，也许需要使用图像编辑器将其剪切成很多个较小的图像。如果由于某些原因，您确实需要在运行时将大图像剪切成小图像的话，可以使用 `com.croftsoft.core.awt.image` 包的 `ImageLib` 类中的静态方法 `crop()`来作这种剪裁。

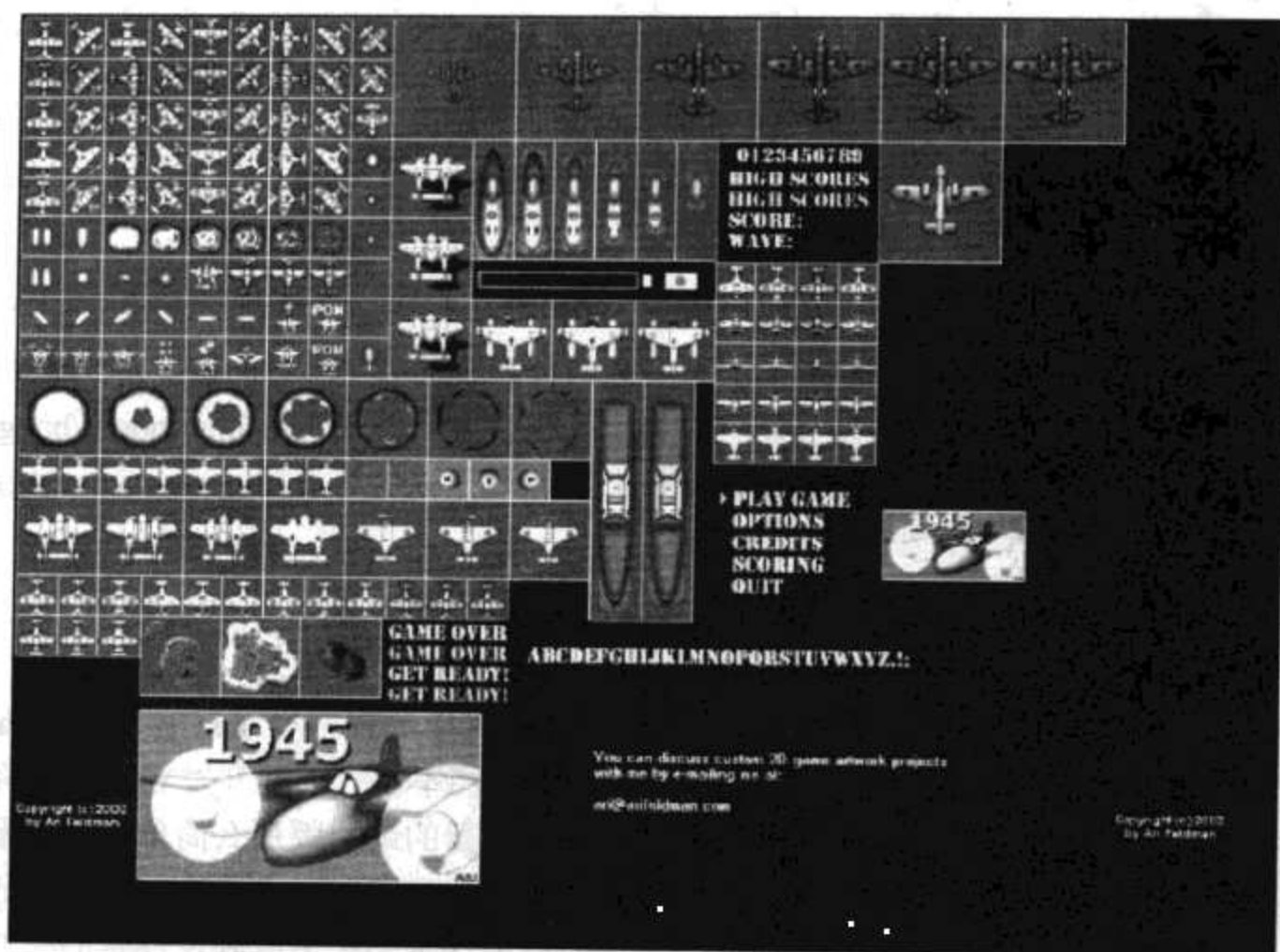


图 2-1 Ari Feldman 的 1945 精灵图片集

在 JAR 时代到来之前，Java applet 开发人员经常需要处理的另外一件让人棘手的事情是，需要编写一种延迟使用图像的代码直到图像完全下载下来。在 Web 的早期，Internet 链接速度很慢，以致于用户可以感觉到正在下载什么东西，并可以决定正在下载的这个东西是否值得进行等待。那些直到图像完全下载下来以后才想使用图像的 applet 开发人员就使用 `java.awt.MediaTracker` 类，去跟踪下载的进度。这一点非常重要，因为那时候的 HTTP 请求更加不可靠，有时候请求下载图像可能会完全失败。如果游戏的开发者没有意识到这一点，那么就可能由于较差的 Internet 链接，出现一些作弊行为，敌人精灵采用隐身术，这样游戏的玩家就可能在与敌人精灵对抗的枪战游戏中屡遭失败。

幸运的是，当在与代码一样的 JAR 文件中包含媒体文件时，已经不必编写跟踪、停止以及下载请求重试逻辑的程序了。在采取这种方法的时候，就应该明白，您必须保证游戏代码能够可靠而且快速地加载图像，而不需要采取其他的预防措施。如果图像下载偶尔失败，那么您对其就不会有别的什么办法，因为也许游戏代码根本还没有下载下来(因为它们是在同一个包中)。

将所有图像和其他资源文件都放置在 JAR 文件内部的一个注意点是，您需要使用 `java.lang.ClassLoader` 类的 `getResource()` 方法来访问这些文件。用于实现这种目的的 `ClassLoader` 必须继承于一个类，这个类与准备访问的资源文件在相同的 JAR 文件中。这是因为从不同的 JAR 文件中加载的类必须要分配它们自己的 `ClassLoader`。`getResource()` 方法的路径参数和文件名参数还必须和与 `ClassLoader` 的实例关联的 JAR 文件的根路径相对应。

```
URL imageURL
    = getClass ( ).getClassLoader ( ).getResource ( imageFilename );

Image image = ImageIO.read ( imageURL );
```


注意, 从 `getResource()` 方法中返回的对象是 `URL` 类的一个实例而不是 `File` 类的一个实例。下面是使用 `URL` 访问数据的另一个示例:

```
URL bangAudioURL = getClass ( ).getClassLoader ( )
    .getResource ( BANG_AUDIO_FILENAME );

AudioClip bangAudioClip = Applet.newAudioClip ( bangAudioURL );
```

其中的另外一个注意点是 `getResource()` 方法的路径和文件名称参数, 虽然与 `JAR` 归档文件的根目录相关联, 但是绝对不能使用反斜杆(/)指示其根目录。例如, `media/dodger/bang.wav` 这样的路径是正确的, 但是 `/media/dodger/bang.wav` 就不正确了。这与使用 `getResource()` 方法从包含 `JAR` 文件的驱动器中拖出文件的方法有些不同。

2.1.4 使用插件升级客户端

`applet` 嵌入到浏览器客户端带来的主要问题之一是, 其中的很多程序只能支持 `Java` 原来的版本, 例如 `Java 1.1.1`。我一直都主张升级客户端程序, 而不使代码和开发者的技能降级。但是, 升级客户端并不像请求用户升级到他们所使用的浏览器的最新版那么简单, 因为最新版中仍然可能会包含一些原来版本的 `Java`。例如, 在写这本书的时候, `Internet Explorer` 就是和 `Java 1.1` 一起发行的, 而 `Netscape` 是和 `Java 1.3` 一起发行的。因为这本书提倡或者说推崇的是 `Java 1.4`, 所以这可能会成为一个问题。

能解决这个问题的是 `Java` 插件, 以前叫 `Activator`。如果浏览器的供应商不去升级嵌入在浏览器中 `Java` 的版本, 这种插件会替您实现这种升级。插件是通过在 `HTML Web` 页面(这种页面中包含有将 `applet` 看作浏览器插件的指令)中添加特殊的标记来实现这种升级的。当浏览器判断它没有所需插件的合适版本时, 它就会提示用户去下载最新的版本。

```
<html>
<body>
<applet
  code="com.croftsoft.apps.collection.CroftSoftCollection.class"
  archive="collection.jar"
  width="600"
  height="400">
</applet>
</body>
</html>
```

上面代码是一个带有 `applet` 标记的简单示例 `Web` 页面。

```
<html>
<body>
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
  classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  codebase = "http://java.sun.com/products/plugin/autodl/jinstall-1_4-
    windows-i586.cab#Version=1,4,0,0"
  WIDTH = "600" HEIGHT = "400" >
  <PARAM NAME = CODE VALUE =
```



```

        "com.croftsoft.apps.collection.CroftSoftCollection.class" >
<PARAM NAME = ARCHIVE VALUE = "collection.jar" >
<PARAM NAME = "type" VALUE = "application/x-java-applet;version=1.4">
<PARAM NAME = "scriptable" VALUE = "false">

<COMMENT>
<EMBED
    type = "application/x-java-applet;version=1.4"
    CODE
        = "com.croftsoft.apps.collection.CroftSoftCollection.class"
    ARCHIVE = "collection.jar"
    WIDTH = "600"
    HEIGHT = "400"
    scriptable = false
    pluginspage
        = "http://java.sun.com/products/plugin/index.html#download">
<NOEMBED>

    </NOEMBED>
</EMBED>
</COMMENT>
</OBJECT>

<!--
<APPLET CODE = "com.croftsoft.apps.collection.CroftSoftCollection.class"
    ARCHIVE = "collection.jar" WIDTH = "600" HEIGHT = "400">

</APPLET>
-->

<!--"END_CONVERTED_APPLET"-->

</body>
</html>

```

上面的代码就是由 Java 插件转换以后页面的大致模样。

执行这种转换的 Java 插件工具被称为 `HtmlConverter`，这个名称还是比较贴切的。这个工具在 Java SDK 安装目录下的 `bin` 子目录中。它既有命令行，又有图形化的用户界面(GUI)。

```
C:\jdk\bin\HtmlConverter.exe -backup original -latest index.html
```

上面的这段代码就是用来转换这个页面的命令。要进行转换的文件名称是 `index.html`。因为 `HtmlConverter` 替换了原始的文件，所以用它去保存 `original` 子目录的备份。在与 Java 的最新版本相同或更新的客户端上允许使用老版本 Java。如果没有使用 `-latest` 选项，代码就会请求用户下载 Java 的特定版本，甚至是在一个较新的版本已经安装以后还是这样。

除了这里提到的以外，Java 插件还有很多特性。在线图书 *Java Plug-in 1.4.2 Developer Guide* 详细地介绍了这些特性²。

² <http://java.sun.com/j2se/1.4.2/docs/guide/plugin/>

2.1.5 了解存在的限制

浏览器 applet 运行在客户机上的一个安全沙箱(sandbox)内, 在其中运行时, 它们不会带来任何伤害。这说明在 applet 能够做的事情上存在着一些限制。例如, 为了缓解用户在网上冲浪的压力, 下载的 applet 对他们的硬盘驱动器没有读和写的访问权限。为了防止 applet 将客户机作为平台策动对 Internet 上其他机器的攻击, applet 只能与从其上下载它的那个服务器的网络建立连接。记住, 所连接的这个服务器必须是从其下载该 applet 的那个服务器——codebase——这个服务器并不总是与为含有嵌入式 applet 的 HTML Web 页面提供服务的那个服务器相同——那个服务器叫 document base。

虽然还有其他的一些附加限制, 但是游戏开发人员最需要专门处理的就是这两个限制。不能够访问硬盘驱动器, 就使 applet 不能保存玩家的最高得分或首选设置。不能够连接除了 applet 服务器以外的其他机器, 就说明游戏 applet 不能直接和其他机器建立连接而进行多玩家网络游戏。

避开这两个限制的方法是在 applet 的 codebase 服务器上执行这些功能。在提示玩家输入用户名和密码以后, applet 就可以向 applet 服务器创建一个网络连接加载用户的数据。然后 applet 就可以创建其他的网络连接在服务器上保存游戏的状态——可以按照玩家需要保存游戏的状态。也可以在游戏进行的过程中, 按照一定的时间间隔定期地保存这些状态数据。在多人玩家网络游戏的情况下, 玩家之间的所有消息都通过中央服务器间接地相互路由到对方。例如, 假如有一个玩家的精灵移动到某一个虚拟空间上时, applet 上其他玩家的视图都是由一个首先从工作 applet 发到 applet 服务器, 然后再从 applet 服务器发送到接收 applet 的消息进行更新的。

applet 联网的问题, 包括利用防火墙隧道技术的 applet-servlet 消息传递功能, 将在本书的最后三章进行详细地介绍。

2.1.6 applet 签名

可以通过对 applet 进行数字签名的方法, 解除 applet 的安全限制。对 applet 进行数字签名就可以让框架验证这个 applet 是不是由您(开发人员)创建的, 而不是由其他人冒充您创建的。它还允许框架检验自从您创建这个 applet 以后, 是不是有人对它进行过修改。通过验证以后, applet 框架(例如浏览器或 Java 插件)就可以立即授予运行 applet 的权限, 而没有任何安全限制。

如果用户批准了这个权限, 签名以后的 applet 就被说成是可信的。那么没有签名的 applet 就只能运行在不可信赖的模式下。这样我信赖的 applet 就是在安全沙箱限制下运行的未签名 applet, 这一点让我觉得混乱。当您认为验证这种安全措施不够坚固的时候, 授予 applet 完全访问您的机器将会是非常愚蠢的。

但是也不要因为这种说法而打击了您对 applet 进行签名的热情。您不应该泄气, 因为还有下面这样的事实: 支持数字签名的 applet 在从一个浏览器到另一个浏览器过程中会发生变化, 并且有时还需要浏览器专用的签名过程。此外, 您可能还需要向受到信赖的证书认证机构缴纳几百美元费用, 这样才能肯定地被标识出来。最后, 您可能还会考虑那些妄想者以及具有隐私意识的用户的反应——当他们被提示决定是否让您的游戏 applet 完全控制他们家里和办公室里的计算机的时候。

有关 applet 签名的更多信息，请参考 *Java Plug-in Developer Guide for J2SE 1.4*³。

2.1.7 缓存 applet

applet 发布的一个主要优点就是用户每次玩您游戏的时候，都会下载最新版本的代码。从另一方面来讲，applet 发布的一个主要缺点也是用户每次玩游戏的时候，都会下载最新版本的游戏代码。这样，您需要做的事情就是为浏览器缓存 applet，仅当这个 applet 有最新可用版本的时候才下载其代码。尽管现在浏览器已经能够在浏览器的会话(session)之间缓存 Web 页面和图像，但是它不能缓存 applet。当用户关闭浏览器窗口的时候，浏览器就会丢弃下载的 applet，这样做也许是考虑安全方面的原因。

过去，我曾经编写过用来实现 applet 缓存功能的定制代码。这其中要涉及到数字签名 applet 的技术和当前一些主要网络浏览器的各种不同安全策略方面一些比较秘密的技术细节。然而，现在就不再需要这样的定制代码了，因为现在的 Java 插件支持 applet 缓存。我不能保证这种特性的实际效果，因为我还没有真正使用过它，只是在文章中看过对它这种功能的介绍。如果您的 applet 很“胖”，而您的用户又是使用较慢的 Internet 链接，或您只是想节省带宽占用费用，Java 插件的 applet 缓存机制确实值得您去试一试。

2.2 部署为可执行 JAR

除了可以进行 applet 发布以外，还可以将 JAR 文件发布为自包含的可执行文件。用户只需要将 JAR 文件复制到桌面或硬盘驱动器的目录上，就可运行程序。如果可执行文件是一个 Swing 应用程序，那么如果用户双击这个 JAR 文件，它就会在一个新窗口中启动程序。如果可执行文件是一个命令行应用程序，那么用户需要输入带有 -jar 选项的 java 命令就可以了。

```
java -jar fraction.jar
java -cp J:\lib com.croftsoft.apps.fraction.FractionAction
```

对玩家来说，双击 JAR 文件或使用带有 -jar 选项的 java 命令行启动游戏，比在命令行中给出带有完全包名称前缀的 classpath 和类名称要容易一些。比较上面的这两个 java 命令，一个是使用 -jar 选项，而另一个没有使用这个选项。

```
public static void main ( String [ ] args )
```

当玩家启动可执行 JAR 文件的时候，java 命令框架就会在一个指定的类中查找并调用静态 main() 方法。这样才会使游戏启动起来。在这种情况下，可以将 main() 方法看作是一个生命周期方法。在这里，我实际对这个概念做了一些延伸。

2.2.1 生成清单文件

JAR 文件通常会包含很多类，其中带有 main() 方法的类不止一个。为了确定哪一个类应该含有要启动的 main() 方法，java 命令要在清单文件中查找 Main-Class 入口。

```
Main-Class: com.croftsoft.apps.fraction.FractionAction
```

³ <http://java.sun.com/j2se/1.4.2/docs/guide/plugin/>

这行代码是清单文件中的一行，但是也可能就是惟一的一行。这行代码是在有 XML 之前，以常在属性文件中使用的标准的名-值对的格式写入的。通过冒号和空格，您可以将类名称的值和属性名称区分开来。

警告：

记住，冒号后面只能跟一个空格。如果您在类名称前使用了两个空格，它就不能运行了。

```
jar -cvfm collection.jar manifest.txt.
```

当使用 jar 命令的时候，JAR 文件一般都会自动地包含一个 META-INF/MANIFEST.MF 形式的空白清单文件。为了包含一个含有 Main-Class 入口的非空清单文件，需要使用 jar 命令的 -m 选项。在上面的示例中，使用 jar 命令明确要包含文本文件 manifest.txt。当 manifest.txt 被包含进来以后，它就会自动地被更名为 MANIFEST.MF，并存储在归档文件的 META-INF 目录中。

```
<jar
  jarfile="collection.jar"
  basedir="jar"
  manifest="bld/apps/collection/manifest.txt"/>
```

在 Ant 构建文件中，可以使用 manifest 属性为要包含的清单文件指定路径。

2.2.2 对不安全性进行保护

可执行 JAR 文件通常都运行在客户端的平台上，并带着一些自由地访问敏感资源(例如硬盘驱动器和默认 Internet 连接)的权限。这可能会使那些具有丰富经验的用户可以很狡猾去玩那些发布为可执行 JAR 的游戏，尤其是当他们不是很清楚发布者的时候。这种用户通常更喜欢玩那些自动限制到一个安全沙箱(例如那些被发布为浏览器 applet 或 Java Web Start 应用程序)的游戏。

```
java -Djava.security.manager -jar collection.jar
```

在使用 java 命令属性 java.security.manager 运行 JAR 文件的时候，可以指定一个要使用的安全性管理器。在上面的命令行示例中，没有给出这个属性的值，因此，所有可能被施加在未签名的 applet(在浏览器中运行)上的约束都被强加在可执行的 JAR 上。如果使用这个属性，也将是安全的。记住，如果它不是被设计成在安全沙箱中运行的话，游戏可能会引发一些 SecurityExceptions 异常，并不能正常运行。

2.3 用 Java Web Start 进行部署

Java Web Start 具有基于浏览器的 applet 发布和持久的桌面应用程序发布的所有优点，而没有这二者的缺点。当用户访问 Web 页面的时候，Web 页面会提示他们下载游戏，以及像运行桌面应用程序一样运行游戏。因为游戏是持久地安装在客户端的，所以用户以后玩游戏时不需要 Internet 链接。一旦您重新修改了 Web 页面上的代码，Java Web Start 就会使用最新版本的代码自动地更新用户客户端的安装程序。用户下载和运行这些游戏时会觉得很舒服，因为它们像 applet 一样运行在安全沙箱以内。与 applet 不同，您不需要服务器端代码在服务器中存储用户

的游戏数据，因为 Java Web Start 的沙箱足够宽松，足以允许您持久地存储少量游戏数据，这种数据的存储形式与浏览器的 cookie 有些相似。

由于这些原因，就我个人来说，Java Web Start 是发布 Java 游戏的一种最好方法。作为游戏玩家，我希望能够玩一些复杂和细化的 Java 游戏，而不需要在每次玩游戏的时候都下载巨大的 JAR 文件。同时，我还担心由那些陌生人编写的、且作为礼物送给我的程序是否会访问我的硬盘驱动器。作为游戏的开发人员，我希望能够编写大型的游戏，并只在我发布新版本的时候，才交付带宽占用费用。另外，我不希望支持服务器集群，只是想保存单个游戏玩家的游戏数据，例如最高分和用户首选设置。Java Web Start 恰恰满足了我的这两个愿望。

为 Windows 操作系统安装 Sun Microsystems 实现的 Java 运行环境(Java Runtime Environment, JRE)的 1.4 版本时，会自动地安装 Java Web Start。对于 Linux 操作系统，也捆绑了 Java 1.4 实现，但是还需要一个单独的手动安装。而 Apple 会在 Macintosh OS X 10.1 操作系统上预安装 Java Web Start。

Java Web Start 是 Java 网络启动协议(Java Network Launching Protocol, JNLP)客户端规范的 Sun Microsystems 实现。因为 Java Web Start 是基于 JNLP 标准的，所以一些与 Java Web Start 兼容的游戏也是运行在由其他供应商实现的 JNLP 客户端框架以内的。这些框架包括开放源代码项目的实现，例如 OpenJNLP。

2.3.1 准备发布文件

为了发布而设置 Java Web Start 应用程序的过程需要上传一个 Web 页面和多个文件(除了代码之外)。尽管 JNLP 规范和 Java Web Start 开发人员指南详细地介绍了这些内容(请参考后面的参考书目)，但是下面的代码会让您对所需要掌握的内容有一个快速的了解。

```
<html>
<body>
<a href="collection.jnlp">Install the CroftSoft Collection</a>
</body>
</html>
```

这段代码是一个简单 HTML Web 页面的示例，该 Web 页面带有一个超连接，当用户单击该超连接的时候，它会下载和安装 Java Web Start 应用程序。

```
<?xml version="1.0" encoding="UTF-8"?>

<jnlp
  spec="1.0+"
  codebase="http://www.croftsoft.com/portfolio/collection/install"
  href="collection.jnlp">

  <information>
    <title>CroftSoft Collection</title>
    <vendor>CroftSoft Inc</vendor>
    <description kind="one-line">Java programs from CroftSoft.</description>
    <description kind="short">
      A collection of Java programs from CroftSoft.
    </description>
```

```

<description kind="tooltip">CroftSoft Collection</description>
<homepage href="http://www.croftsoft.com/portfolio/collection/">
<icon kind="default" href="default_icon.gif"/>
<offline-allowed/>
</information>

<resources>
  <j2se version="1.4+" href="http://java.sun.com/products/autodl/j2se"/>
  <jar href="collection.jar"/>
</resources>

<application-desc mainclass="
com.croftsoft.apps.collection.CroftSoftCollection"/>

</jnlp>

```

上面的代码是由 Web 页面引用的 collection.jnlp 文件的全部内容。像 Ant 一样, JNLP 使用 XML 作为其配置说明。如果客户机上还没有 Java 的 1.4 或更新版本的时候, 这个 XML 文件会告诉 JNLP 客户端(例如 Java Web Start)去提示用户安装 Java 的最新版本。然后它会启动归档文件 collection.jar 中的代码。从下一行直到最后一行指定了包含用于启动程序的 main()方法的附加类。

```
AddType application/x-java-jnlp-file .jnlp
```

除了包含程序和进行一次安装所需要的一个桌面图标的 JAR 文件以外, 可能还需要在发布目录中包含一个附加文件。前面给出的这个文件被命名为.htaccess, 如果使用了 Web 服务器(还没有配置为自动地将带有.jnlp 扩展名的文件与 MIME 类型的 type application/x-java-jnlp-file 文件关联), 那么可能需要使用这个文件。如果 Web 页面服务提供商不愿意重新配置它的 MIME 类型设置, 您可以将这个文件放在您的发布目录中, 作为一个覆盖设置的说明——如果您使用的是 Apache Web Server 的话。这是很必要的, 因为浏览器使用由 Web 服务器发送的 MIME 类型, 来确定要使用哪一个插件——在这里是 Java Web Start 或另一个 JNLP 兼容客户端。

警告:

忘记重新配置 Web 服务器, 使得它将.jnlp 文件扩展名与 MIME 类型关联起来, 是我和学生们经常遭遇的一个问题。如果您在游戏中单击了超链接, 浏览器就会将.jnlp 文件显示为文本文件而不是启动 JNLP 客户端, 这就说明您也忘了重新配置 Web 服务器。请尝试上传.htaccess, 并看看这样是否就解决了这个问题。

```

<security>
  <all-permissions/>
</security>

```

如果确实需要那样做的话, 也可以运行 Java Web Start 应用程序, 而不要被安全沙箱所限制。如果 JAR 文件是经过数字签名的, 可以在.jnlp 文件中插入像前面那样的 XML 代码。这段代码使游戏可以读取或写入所有的硬盘驱动器, 并可以使所有的 Internet 套接字连接到任何地方。

```

<signjar
  jar="mygame.jar"

```



```
storepass="${password}"
alias="myself"
keystore="myKeystore"
keypass="${password}" />
```

可以使用像这段代码一样的 Ant 构建指令，自动地将 JAR 文件进行数字签名。这样做的目的是，确保在用户做出相信运行您的代码不会在他们的系统上产生什么问题的判断以前，您可以伪装成任何人。如果您不想花钱从受到信赖的第三方那里取得认证的证书，也可以自签名您的 JAR 文件。在自签名 JAR 文件的时候，JNLP 客户端会警告用户，它不能鉴别发布者的身份，并警告用户有安全风险。然后它就提示用户取消或继续。

其他有关信息，请参考 *Java Network Launching Protocol & API Specification* 和 *Java Web Start 1.4.2 Developer Guide*⁴。

2.3.2 访问默认浏览器

Java 桌面应用程序有时还需要向用户提供新闻、文档和注册表单的在线 Web 页面。尽管 Java 现在提供了使用 `javax.swing.JEditorPane` 类显示 HTML 这种有限的能力，但是 Web 浏览器，例如 Netscape 或 Internet Explorer，还具有一些辅助的高级功能，例如支持 HTML 脚本语言和多种不同的多媒体格式。这一节主要介绍怎样让 Java 桌面应用程序可以在应用程序的客户机平台以外默认的浏览器上，以一种平台独立的方式启动。

1. showDocument()方法

Java 一直都支持 applet 通过接口 `java.applet.AppletContext` 的 `showDocument()` 方法控制它的浏览器容器的这种能力。遗憾的是，Java 桌面应用程序与 applet 相反，它不能访问 `AppletContext` 的实例。JNLP API 提供了一个接口 `javax.jnlp.BasicService`——这个接口为 JNLP 桌面应用程序提供了一个相类似的方法。

2. 分离可选包

如果在客户机平台上，可以使用 JNLP API 实现，那么 `BasicService` 类的 `showDocument()` 方法就会在客户机平台上使用平台无关的代码启动默认的 Web 浏览器。但是，我们通常都会希望能够以在没有安装 JNLP 库的环境中仍然能够运行的那种方式编写游戏代码。理想的情况下，编译以后的游戏代码应该能够以浏览器 applet、可执行 JAR 和 JNLP 的 Java Web Start 应用程序这 3 种方式运行而不需要进行任何修改。如果 JNLP 类库在客户机上可用的话，就可以通过包含加载 JNLP 类库代码的方法实现这个功能，如果不可用的话，程序也能很好地继续运行。

```
private static JnlpServices createJnlpServices ( )
///////////////////////////////////////////////////
{
    try
    {
        return ( JnlpServices )
            Class.forName ( JnlpServices.IMPL_CLASS_NAME ).newInstance ( );
    }
}
```

⁴ <http://java.sun.com/products/javawebstart/download-spec.html>, <http://java.sun.com/j2se/1.4.2/docs/guide/jws/developersguide/contents.html>

```

    }
    catch ( Exception ex )
    {
        return null;
    }
    catch ( NoClassDefFoundError er )
    {
        return null;
    }
}

```

这里使用的技巧是使用动态类加载。也可以使用反射来实现这种技巧，但是我发现，一般的规则是，最好还是使用定制的接口和动态类加载。首先要将静态链接到可选包库 `javax.jnlp` 的代码分离出来，然后再尝试动态加载使用 `Class.forName().newInstance()` 的代码。如果还没有安装 JNLP，这种尝试就会产生一个 `NoClassDefFoundError` 异常，可以使用这个方法优雅地捕获和处理这个异常。上面的代码就是 `com.croftsoft.core.jnlp` 包的 `JnlpLib` 类中的代码，这段代码就是专门完成这种功能的。

```

package com.croftsoft.core.jnlp;

import java.io.*;
import java.net.*;

public interface JnlpServices
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

    public static final String IMPL_CLASS_NAME
        = "com.croftsoft.core.jnlp.JnlpServicesImpl";

    [...]

    public boolean showDocument ( URL url )
        throws UnsupportedOperationException;
}

```

在这里，`JnlpServices` 是没有和 `javax.jnlp` 包建立静态链接的 `com.croftsoft.core.jnlp` 包的一个接口。`IMPL_CLASS_NAME` 是一个类 `JnlpServicesImpl` 的名称。这个类提供了不需要与可选包进行静态链接的 `JnlpServices` 接口的一个具体实现。对定制接口的引用允许代码处理可选包库，而无需使用静态链接，也不需要反射。

提示：

在讲授 Java 游戏编程课程的过程中，我对我的学生做过这样的说明——在 Ant 构建文件中，忘记显式地命名动态链接的类是导致失误的最常见原因之一。这种问题可能很难进行调试，因为它在编译时并不会有什么问题，甚至也不会产生运行时错误。这种调用代码可能会忽略运行时错误，而在假定 `JnlpServicesImpl` 不能动态加载的情况下继续运行，因为游戏不是运行在 JNLP 客户端容器内。但是这个问题的实质的确是 `JnlpServicesImpl` 不能被加载，因为它还没有被编译和包含在 JAR 文件中。


```

public static final JnlpServices JNLP_SERVICES
    = createJnlpServices ( );

[...]

public static boolean showDocument ( URL url )
    throws UnsupportedOperationException
    //////////////////////////////////////
{
    check ( );

    return JNLP_SERVICES.showDocument ( url );
}

[...]

private static void check ( )
    throws UnsupportedOperationException
    //////////////////////////////////////
{
    if ( JNLP_SERVICES == null )
    {
        throw new UnsupportedOperationException ( );
    }
}

```

如果能被动态加载的话，类 JnlpLib 中的静态方法 showDocument() 就使用实现类。如果不能，它就抛出 UnsupportedOperationException 异常，调用游戏的代码可以从容地捕获和处理这个异常。

```

package com.croftsoft.core.jnlp;

import java.io.*;
import java.net.*;

import javax.jnlp.*;

[...]

public final class JnlpServicesImpl
    implements JnlpServices
    //////////////////////////////////////
    //////////////////////////////////////
{

    [...]

    public boolean showDocument ( URL url )
        throws UnsupportedOperationException
        //////////////////////////////////////
    {
        try

```

```

{
    BasicService basicService = ( BasicService )
        ServiceManager.lookup ( "javax.jnlp.BasicService" );

    return basicService.showDocument ( url );
}
catch ( UnavailableServiceException ex )
{
    throw ( UnsupportedOperationException )
        new UnsupportedOperationException ( ).initCause ( ex );
}
}

```

使用 `AppletContext` 中 `showDocument()` 方法的时候，就会使用 `applet` 实例自己的 `getAppletContext()` 方法来获取 `AppletContext` 的一个实例。但是，也会使用 `javax.jnlp.ServiceManager` 类的静态方法 `lookup()` 来获取 `BasicService` 接口的一个实例。

```

public static Object lookup ( String name )
    throws UnavailableServiceException;

```

如果在 JNLP 客户端不能使用 `BasicService` 的实现，那么 `ServiceManager` 就会抛出 `UnavailableServiceException` 异常。如果被抛出的话，`JnlpServicesImpl` 就将可选包 `javax.jnlp` 中的这个异常转换为内核包 `java.lang` 中的 `UnsupportedOperationException` 异常。注意，通过调用代码，使用 `Throwable.initCause()` 方法将原异常与新异常联系在一起。

2.3.3 使用反射进行动态链接

另一种选择是，使用反射动态加载 JNLP 类，正如下面从 `com.croftsoft.core.jnlp` 包的 `JnlpProxy` 类中取出的这段代码这样。

```

private static Object getBasicServiceObject ( )
///////////////////////////////////////////////////
{
    try
    {
        Class serviceManagerClass
            = Class.forName ( "javax.jnlp.ServiceManager" );

        Method lookupMethod = serviceManagerClass.getMethod (
            "lookup", new Class [ ] { String.class } );

        return lookupMethod.invoke (
            null, new Object [ ] { "javax.jnlp.BasicService" } );
    }
    catch ( Exception ex )
    {
        return null;
    }
}

```


类 `Method` 是内核包 `java.lang.reflect` 中的一个类。

```
public static boolean showDocument ( URL url )
///////////////////////////////////////////////////////////////////
{
    if ( basicServiceObject == null )
    {
        return false;
    }

    try
    {
        Method method = basicServiceClass.getMethod (
            "showDocument", new Class [ ] { URL.class } );

        Boolean resultBoolean = ( Boolean ) method.invoke (
            basicServiceObject, new Object [ ] { url } );

        return resultBoolean.booleanValue ( );
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );

        throw new RuntimeException ( ex.getMessage ( ) );
    }
}
```

通过保存一个对获取的 `BasicService` 实例的静态引用，就可以在静态的 `showDocument()` 方法中使用它。如果客户端不支持 JNLP，这个示例方法就会返回 `false`，允许调用代码用一个替代的行为来响应。

在我个人看来，使用定制接口的动态链接代替反射能使代码更容易理解。记住了这一点的话，几乎就不需要使用反射技术了。

2.4 将多个 applet 部署为一个 applet

这一节将介绍一些可以针对自己的游戏进行修改和使用的示例框架代码。可以将它发布为嵌入在 Web 页面中的 applet，也可以将它发布为可执行 JAR 文件或 Java Web Start 应用程序。主类 `MultiApplet` 被构造为一种超级 applet，这种 applet 可以包含任意数目的其他 applet。如图 2-2 所示，`MultiApplet` 允许玩家滚动嵌入在这一个 applet 中的游戏列表。每次可以玩全部游戏，而不需要重新启动程序或重新加载页面。`MultiApplet` 也提供了用于显示游戏新闻和文档的一个面板。

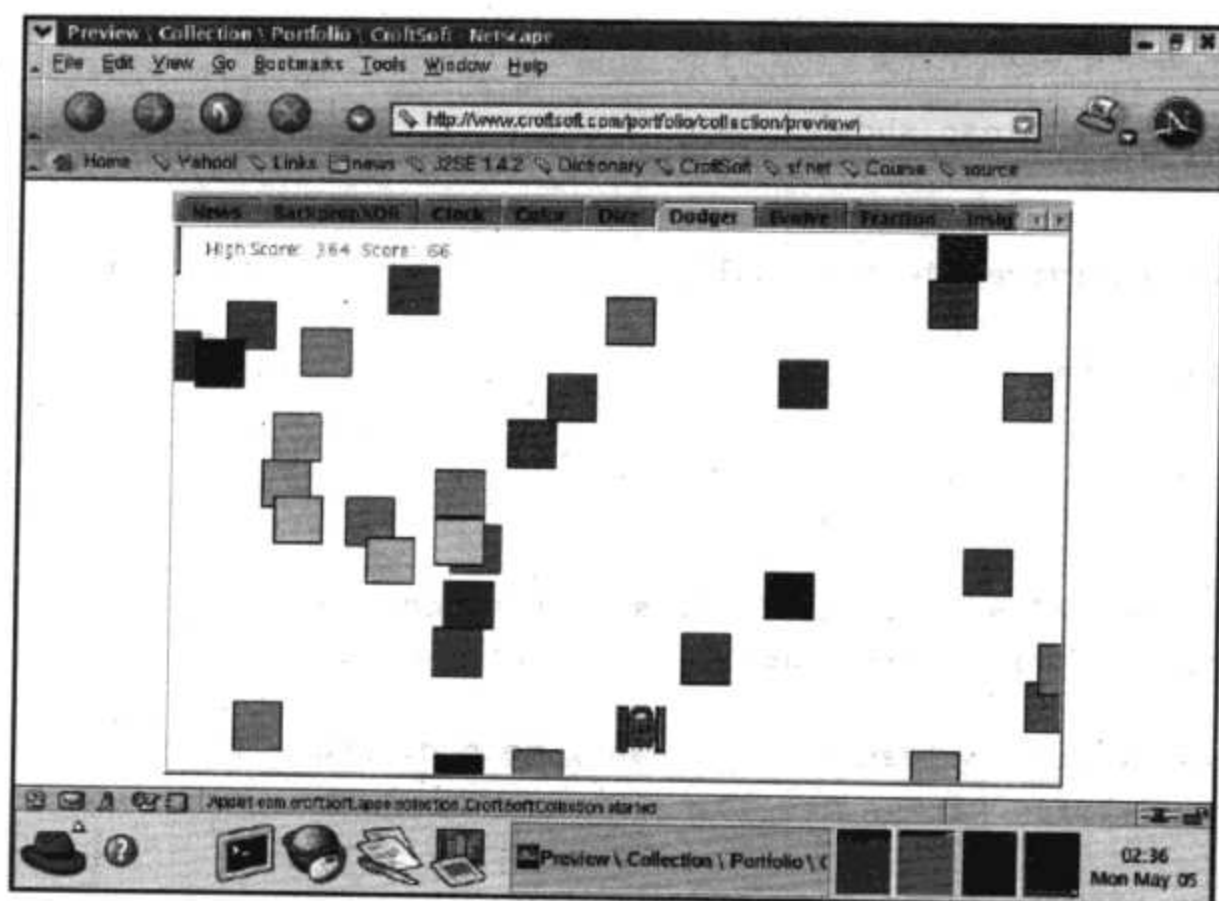


图 2-2 MultiApplet 示例

2.4.1 MultiAppletStub

容器首次初始化 Applet 实例的时候，它就调用 `setStub()` 方法来传递 `AppletStub` 接口的一个实例。Applet 使用 `AppletStub` 访问 `AppletContext` 和其他一些属性。

```
package com.croftsoft.core.gui.multi;

import java.applet.*;
import java.net.*;

import com.croftsoft.core.lang.NullArgumentException;

[...]
```

```
public final class MultiAppletStub
    implements AppletStub
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////
    private final Applet parentApplet;
    private boolean active;

    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    public MultiAppletStub ( Applet parentApplet )
    {
        NullArgumentException.check ( this.parentApplet = parentApplet );
    }
}
```


MultiApplet 是一个包含其他 applet 的 applet。它也是另外一个 applet 容器(通常是 Web 浏览器, 如图 2-2 所示)的容器。作为一个容器框架而不是一个被包容者角色的 MultiApplet, 必须对它的子 applet 提供 AppletStub 实例。MultiAppletStub 为达到这个目的实现了 AppletStub 接口。

```
public void appletResize (
    int width,
    int height )
    //////////////////////////////////////
{
    parentApplet.resize ( width, height );
}

public AppletContext getAppletContext ( )
    //////////////////////////////////////
{
    return parentApplet.getAppletContext ( );
}

public URL getCodeBase ( )
    //////////////////////////////////////
{
    return parentApplet.getCodeBase ( );
}

public URL getDocumentBase ( )
    //////////////////////////////////////
{
    return parentApplet.getDocumentBase ( );
}

public String getParameter ( String name )
    //////////////////////////////////////
{
    return parentApplet.getParameter ( name );
}
```

MultiAppletStub 通过对 parentApplet(一般为 MultiApplet 的一个实例)的委托调用, 为子 applet 提供容器服务。例如, 当一个子游戏 applet 获取 AppletContext 去调用像 showDocument() 这样的方法时, 它就使用父 applet 从浏览器中获取的同一个 AppletContext 实例。

```
public boolean isActive ( )
    //////////////////////////////////////
{
    return active;
}

public void setActive ( boolean active )
    //////////////////////////////////////
{
    this.active = active;
}
```

AppletStub 接口定义了 accessor 方法 isActive(), 这样 applet 的运行状态就确定了下来。MultiAppletStub 定义了相应的 mutator 方法。因为当父 applet 处于活动状态时, 子 applet 可能并不处于活动状态, 所以这个属性并没有共享。

2.4.2 MultiAppletNews

使用 com.croftsoft.core.gui.multi 包中的 MultiAppletNews 类来显示游戏的在线新闻和文档, 如图 2-3 所示。

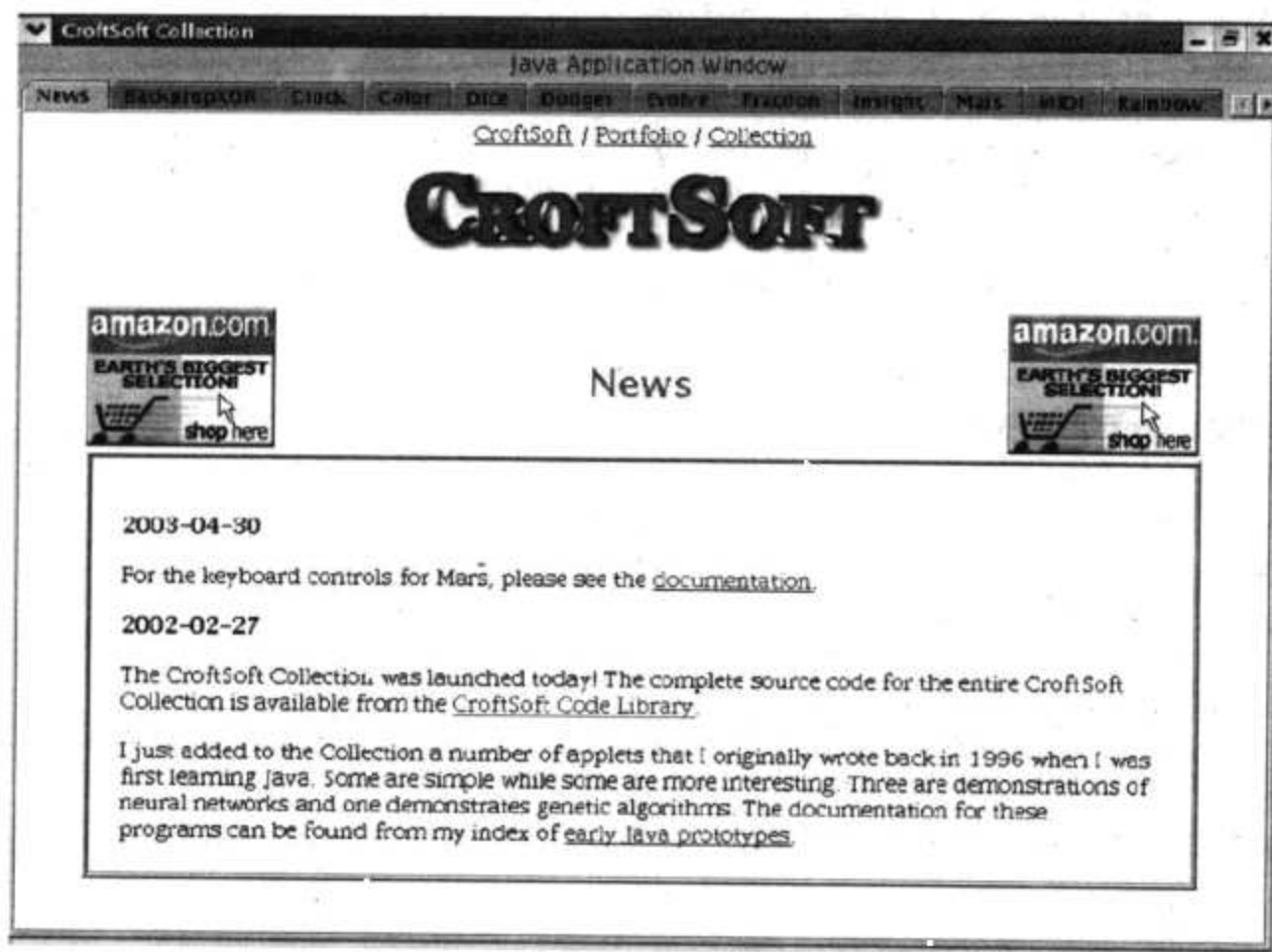


图 2-3 MultiAppletNews

```
package com.croftsoft.core.gui.multi;
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.jnlp.JnlpLib;

public final class MultiAppletNews
    extends JPanel
```



```

////////////////////////////////////
////////////////////////////////////
{

```

MultiAppletNews 是从 **JPanel** 而不是从 **JApplet** 继承来的，这样继承的原因是由于它不是动画，所以也不需要生命周期方法。

```

private static final String DEFAULT_NEWS_HTML
    = "<html><body><pre>"
    + CroftSoftConstants.DEFAULT_ATTRIBUTION_NOTICE
    + "</pre></body></html>";

```

```
//
```

```
private final AppletContext appletContext;
```

```
private final JEditorPane    jEditorPane;
```

如果框架是以浏览器中的 **applet** 的形式运行的，而不是以桌面应用程序的形式运行的，那么当用户单击超连接的时候，**MultiAppletNews** 就使用它的 **appletContext** 重新打开一个浏览器窗口。**jEditorPane** 显示初始的 Web 页面。如果没有指定初始的 HTML，它就会使用 **DEFAULT_NEWS_HTML** 页面。

```

public MultiAppletNews (
    String newsHTML,
    String newsPage,
    Applet applet )
{
    //////////////////////////////////////
    super ( new BorderLayout ( ) );

    AppletContext appletContext = null;

    try
    {
        appletContext = applet.getAppletContext ( );
    }
    catch ( Exception ex )
    {
    }

    this.appletContext = appletContext;

```

在构造函数内部，**appletContext** 是从 **applet** 中获取的。如果 **applet** 为空，或 **applet** 不是运行在一个 **applet** 容器中，这就可能会引发异常。由于实例变量 **appletContext** 声明为 **final**，所以赋值是通过一个临时方法变量实现的。

```

if ( newsHTML == null )
{
    if ( newsPage != null )
    {

```

```

newsHTML
    = "<html><body>"
    + "Loading "
    + newsPage
    + "... "
    + "</body></html>";
}
else
{
    newsHTML = DEFAULT_NEWS_HTML;
}
}

```

下载 newsPage 的时候，首先会显示 newsHTML。如果 newsPage 为空或下载失败，就将继续显示 newsHTML。前面的代码为 newsHTML 提供了默认的值。

```

jEditorPane = new JEditorPane ( "text/html", newsHTML );

jEditorPane.setEditable ( false );

jEditorPane.setCaretPosition ( 0 );

jEditorPane.addHyperlinkListener (
    new HyperlinkListener ( )
    {
        public void hyperlinkUpdate ( HyperlinkEvent hyperlinkEvent )
        {
            processHyperlinkEvent ( hyperlinkEvent );
        }
    } );

add ( new JScrollPane ( jEditorPane ), BorderLayout.CENTER );

```

上面的代码使用内核包 javax.swing 中 JEditorPane 类的实例在多 applet 框架中的第一个面板中显示初始 Web 页面。(^)符号的位置被设为 0，这样当第一次加载页面的时候，就会显示该页面的顶部，而不是滚动到底部。添加一个 HyperlinkListener 之后，它就可以截获超文本链接上的鼠标单击事件，并显示恰当的 Web 页面。有关 JEditorPane 和 HyperlinkListener 的使用指南，我建议您参考 Kim Topley 所著的 *Core Swing Advanced Programming* (Upper Saddle River, NJ: Prentice Hall PTR, 2000) 的第 4 章。

```

if ( newsPage != null )
{
    try
    {
        final URL newsURL = new URL ( newsPage );

        new Thread (
            new Runnable ( )
            {
                public void run ( )
            }
        ).start();
    }
    catch ( Exception e )
    {
        // Handle exception
    }
}

```



```

    {
        try
        {
            jEditorPane.setPage ( newsURL );
        }
        catch ( Exception ex )
        {
            ex.printStackTrace ( );
        }
    }
    } ).start ( );
}
catch ( MalformedURLException ex )
{
    ex.printStackTrace ( );
}
}
}

```

在初始化的时候，MultiAppletNews 试着在 newsURL 上下载 Web 页面。这是在一个独立的线程中启动的，因为如果存在网络问题，它将会被无限地延迟下去。

```

private void processHyperlinkEvent ( HyperlinkEvent hyperlinkEvent )
///////////////////////////////////////////////////
{
    try
    {
        if ( hyperlinkEvent.getEventType ( )
            == HyperlinkEvent.EventType.ACTIVATED )
        {
            if ( hyperlinkEvent instanceof HTMLFrameHyperlinkEvent )
            {
                HTMLDocument htmlDocument
                    = ( HTMLDocument ) jEditorPane.getDocument ( );

                htmlDocument.processHTMLFrameHyperlinkEvent (
                    ( HTMLFrameHyperlinkEvent ) hyperlinkEvent );
            }
            else
            {
                URL url = hyperlinkEvent.getURL ( );

                if ( appletContext != null )
                {
                    appletContext.showDocument ( url, "_blank" );
                }
                else
                {
                    try
                    {
                        Jnlplib.showDocument ( url );
                    }
                    catch ( UnsupportedOperationException ex )

```

```

        {
            jEditorPane.setPage ( url );
        }
    }
}
}
catch ( Exception ex )
{
    ex.printStackTrace ( );
}
}

```

单击超连接时会产生一个 `HyperlinkEvent` 事件。目标 `url` 可以从 `HyperlinkEvent` 事件中获取并显示。可以尝试使用 3 种不同的机制来显示 Web 页面。如果 `AppletContext` 可用，代码就要使用 `AppletContext` 在一个新的浏览器窗口中显示该 URL。否则，它就尝试使用 JNLP 启动一个外部浏览器。如果这也会失败，代码就使用 `JEditorPane` 来显示 Web 页面。注意，与大多数浏览器相比，`JEditorPane` 显示 Web 页面的能力还是比较有限的。

2.4.3 Lifecycle

包 `com.croftsoft.core.lang.lifecycle` 中的接口 `Lifecycle` 定义了 `Applet` 的 4 种生命周期方法：`init()`、`start()`、`stop()` 和 `destroy()`。这个接口的目的是允许我们构建一个用于操纵生命周期对象的框架，而不需要再从超类 `Aplet` 中扩展这些生命周期对象。例如，我们可以拥有这样的一个非可视组件，它必须要进行初始化、启动、停止和销毁，而不需要去扩展 `Applet` 或它的超类 `Panel`。

我们可以拥有一个这样的框架：需要在启动的时候初始化多个对象，并在关闭时销毁这些对象。可以用一个 `Lifecycle` 实例数组，向框架传递多个对象，这样一个普通的例程就可以初始化和销毁这些对象。但是，这样多少会有些让人不太满意，因为 `start()` 和 `stop()` 就成了多余，而且还需要将它们作为空方法体去实现。

我逐渐得出了这样的一个结论：最好的解决办法是为每一个方法定义一个接口，然后再使用多接口继承扩展这些接口。在 `com.croftsoft.core.lang.lifecycle` 包中，也可以找到 `Initializable`、`Startable`、`Stoppable` 和 `Destroyable` 等这些接口，每一个接口都定义了单独的生命周期方法。接口 `Commissionable` 扩展了 `Initializable` 和 `Destroyable`。接口 `Resumable` 扩展了 `Startable` 和 `Stoppable`。接口 `Lifecycle` 通过直接扩展 `Commissionable` 和 `Resumable` 接口，间接地扩展了 `Initializable`、`Startable`、`Stoppable` 和 `Destroyable` 接口。

在同一个包中的类 `LifecycleLib` 还包含了一个静态方法库，这个库用来操纵那些实现生命周期接口的对象。例如，静态方法 `destroy()` 用一个 `Destroyable` 实例数组作为其参数，这些 `Destroyable` 实例依次调用 `destroy()` 方法，捕获并报告在转移到下一个数组元素之前引发的所有异常。

2.4.4 LifecycleWindowListener

对作为 Swing 桌面应用程序独立于像浏览器这样的容器进行运行的示例框架而言，它就必须能够将窗口事件传送到调用游戏的适当生命周期方法中。例如，当窗口首次被激活的时候，

它就应该调用 `init()` 和 `start()` 方法启动动画。当 Windows 窗口最小化的时候，它就调用 `stop()` 方法挂起动画，关闭窗口的时候，就调用 `destroy()` 方法。

```
package com.croftsoft.core.gui;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.croftsoft.core.awt.image.ImageLib;
import com.croftsoft.core.gui.FullScreenToggler;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.lifecycle.AppletLifecycle;
import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.lang.lifecycle.LifecycleLib;

public final class LifecycleWindowListener
    implements WindowListener
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    private final Lifecycle [ ] lifecycles;

    private final String      shutdownConfirmationPrompt;

    private final String      shutdownConfirmationTitle;

    //

    private boolean initialized;

    [...]

    public LifecycleWindowListener (
        Lifecycle [ ] lifecycles,
        String      shutdownConfirmationPrompt,
        String      shutdownConfirmationTitle )
    {
        this.lifecycles = lifecycles;

        this.shutdownConfirmationPrompt = shutdownConfirmationPrompt;

        this.shutdownConfirmationTitle = shutdownConfirmationTitle;
    }

    [...]

    public void windowActivated ( WindowEvent windowEvent )
    {
        ///////////////////////////////////////////////////
        ///////////////////////////////////////////////////
    }
}
```

```

{
    if ( !initialized )
    {
        LifecycleLib.init ( lifecycles );

        initialized = true;
    }

    LifecycleLib.start ( lifecycles );
}

[...]

public void windowClosing ( WindowEvent windowEvent )
///////////////////////////////////////////////////////////////////
{
    Window window = windowEvent.getWindow ( );

    if ( shutdownConfirmationPrompt != null )
    {
        int confirm = JOptionPane.showOptionDialog ( window,
            shutdownConfirmationPrompt,
            shutdownConfirmationTitle != null
                ? shutdownConfirmationTitle : shutdownConfirmationPrompt,
            JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE,
            null, null, null );

        if ( confirm != JOptionPane.YES_OPTION )
        {
            return;
        }
    }

    window.hide ( );

    if ( shutdownConfirmationPrompt == null )
    {
        LifecycleLib.stop ( lifecycles );
    }
    LifecycleLib.destroy ( lifecycles );

    window.dispose ( );

    System.exit ( 0 );
}

public void windowDeactivated ( WindowEvent windowEvent )
///////////////////////////////////////////////////////////////////
{
    LifecycleLib.stop ( lifecycles );
}

[...]

```


包 `com.croftsoft.core.gui` 中的类 `LifecycleWindowListener` 是包 `java.awt.event` 中抽象类 `WindowListener` 的一个具体实现。通过给它传递一组实现了 `Lifecycle` 接口的对象的方法可以使用这个类。当窗口首次被激活的时候，它就调用 `Lifecycle` 对象的 `init()` 方法。它每次也都调用 `start()` 方法。当窗口不激活的时候，它就调用 `stop()` 方法。这意味着，当用户单击窗口或将窗口最大化的时候，游戏动画就会重启，而当用户单击窗口以外的地方或将窗口最小化的时候，游戏的动画就会挂起。

当用户单击关闭窗口图标的时候，就会在顶部显示一个确认窗口。这会使父窗口处于不激活状态，而这又导致了对游戏对象 `stop()` 方法的一个调用，从而会挂起游戏动画。如果用户接着决定继续关闭游戏窗口，在退出游戏之前，就会调用 `destroy()` 方法。而如果用户决定取消关闭窗口命令，确认窗口就会消失。而这就重新激活了父窗口，结果就产生了对游戏对象的 `start()` 方法的调用，从而重新启动游戏动画。

```
public static void launchFrameAsDesktopApp (
    JFrame      jFrame,
    final       Lifecycle [ ] lifecycles,
    Dimension    frameSize,
    String       shutdownConfirmationPrompt )
//////////
{
    NullPointerException.check ( jFrame );

    jFrame.setDefaultCloseOperation (
        WindowConstants.DO_NOTHING_ON_CLOSE );

    jFrame.addWindowListener ( new LifecycleWindowListener (
        lifecycles, shutdownConfirmationPrompt ) );

    if ( frameSize != null )
    {
        WindowLib.centerOnScreen ( jFrame, frameSize );
    }
    else
    {
        WindowLib.centerOnScreen ( jFrame, 0.8 );
    }

    jFrame.show ( );
}
```

静态方法 `launchFrameAsDesktopApp()` 在同一个类的内部是有效的。它向帧中添加 `LifecycleWindowListener` 的一个新实例，然后再调用该实例的 `show()` 方法。通过显示窗口来激活这个实例，这样就调用了游戏对象的生命周期方法 `init()` 和 `start()`。 `Lifecycle` 数组构造函数参数传递游戏对象。如果没有指定 `frameSize`，它默认的值就是屏幕大小的 80%。

```
public static void main ( String [ ] args )
//////////
{
    launchFrameAsDesktopApp (
```

```

new JFrame ( "Test" ),
new Lifecycle [ ] {
    new Lifecycle ( )
    {
        public void init ( ) { System.out.println ( "init" ); }
        public void start ( ) { System.out.println ( "start" ); }
        public void stop ( ) { System.out.println ( "stop" ); }
        public void destroy ( ) { System.out.println ( "destroy" ); }
    },
null, // frameSize
"Exit Test?" );
}

```

类 `LifecycleWindowListener` 中包含一个静态 `main()` 方法，通过从命令行启动的方式，就可以使用这个方法测试和演示它的功能。在操纵包含测试程序的窗口时，生命周期方法就会在标准输出中显示出调试信息。

2.4.5 MultiApplet

```

package com.croftsoft.core.gui.multi;

[...]

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.awt.image.ImageLib;
import com.croftsoft.core.gui.FullScreenToggler;
import com.croftsoft.core.gui.LifecycleWindowListener;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.Pair;
import com.croftsoft.core.lang.lifecycle.Lifecycle;

public class MultiApplet
    extends JApplet
    implements Lifecycle

    //////////////////////////////////////
    //////////////////////////////////////
{

```

主类 `MultiApplet` 扩展了 `Applet` 类的 Swing 版——`JApplet`。它从 `CroftSoft` 的可重用代码库中导入了多个其他类。它实现了带有 `init()`、`start()`、`stop()` 和 `destroy()` 方法的 `Lifecycle` 接口，因此它可以集成到一个普通的 `Lifecycle` 框架容器中。

```

public static final String DEFAULT_NEWS_NAME = "News";

//

private final String appletInfo;

private final Pair [ ] appletPairs;

```



```
private final String  newsName;

private final String  newsHTML;

private final String  newsPage;
```

调用 `getAppletInfo()` 方法的时候，就会返回实例变量 `appletInfo`。变量 `DEFAULT_NEWS_NAME`、`newsName`、`newsHTML` 和 `newsPage` 用于创建 `MultiAppletNews` 的实例。

`Pair` 是包 `com.croftsoft.core.lang` 中的一个便利(*convenience*)类，我用它将 `String` 对象的一个相关配对 (一般都是名-值对) 结合在一起。之所以喜欢使用 `Pair` 数组而不是两个单独的 `String` 数组，是因为这样就再不必担心两个 `String` 数组会意外地变得不等长了。在 `MultiApplet` 类中使用“名”来存储要显示在 `JTabbedPane` 顶部的游戏 `applet` 的简写名称。“值”就是相应游戏 `applet` 的类名称。

```
private JTabbedPane    jTabbedPane;

private Component      appletComponent;

private boolean        isStarted;

private int            index;

private MultiAppletStub multiAppletStub;
```

这里的 `JApplet` 子类维护了对 `JTabbedPane` Swing 组件的一个实例引用。`JTabbedPane` 用于包含游戏的 `applet`。当用户单击标签的时候，就会加载一个不同的游戏。

`appletComponent` 是正在加载游戏的一个引用。注意，它是 `Component` 的一个实例，而不是 `Applet` 的一个实例。这里的布尔标志 `isStarted` 提供了生命周期的状态信息。变量 `index` 指示当前选中的是哪一个标签。`MultiAppletStub` 实例用于向游戏 `applet` 传播 `AppletContext`。

```
public static void main ( String [ ] args )
///////////////////////////////////////////////////
{
    launch (
        CroftSoftConstants.DEFAULT_APPLET_INFO,
        new Pair [ ] {
            new Pair ( "Applet1", "javax.swing.JApplet" ),
            new Pair ( "Applet2", "javax.swing.JApplet" ) },
        DEFAULT_NEWS_NAME,
        ( String ) null,
        CroftSoftConstants.HOME_PAGE,
        "CroftSoft MultiApplet",
        CroftSoftConstants.FRAME_ICON_FILENAME,
        MultiApplet.class.getClassLoader ( ),
        ( Dimension ) null,
        "Close CroftSoft MultiApplet?" );
}
```

也可以从命令行中使用静态方法 `main()` 来测试 `MultiApplet`。它将测试数据传递到下面的 `launch()` 方法中。

```
public static void launch (
    String      appletInfo,
    Pair [ ]    appletPairs,
    String      newsName,
    String      newsHTML,
    String      newsPage,
    String      frameTitle,
    String      frameIconFilename,
    ClassLoader frameIconClassLoader,
    Dimension   frameSize,
    String      shutdownConfirmationPrompt )
////////////////////////////////////
{
    JFrame jFrame = new JFrame ( frameTitle );

    try
    {
        Image iconImage = ImageLib.loadBufferedImage (
            frameIconFilename, frameIconClassLoader );

        if ( iconImage != null )
        {
            jFrame.setIconImage ( iconImage );
        }
    }
    catch ( Exception ex )
    {
    }
}
```

当没有被作为 `applet` 嵌入到 Web 页面上时，静态方法 `launch()` 就会启动程序。这允许它能够以 JAR 桌面应用程序的形式或以 Java Web Start 应用程序的形式运行。它首先创建一个新帧，并设置该帧图标图像，这个图像就是在此帧左上角的一个小图片。如果它找不到这个图像，它就会忽略它并继续向下运行。

```
MultiApplet multiApplet = new MultiApplet (
    appletInfo, appletPairs, newsName, newsHTML, newsPage );
jFrame.setContentPane ( multiApplet );

FullScreenToggler.monitor ( jFrame );

LifecycleWindowListener.launchFrameAsDesktopApp (
    jFrame,
    new Lifecycle [ ] { multiApplet },
    frameSize,
    shutdownConfirmationPrompt );
}
```

然后，它就创建 `MultiApplet` 的一个实例，并将它设为帧的目录页。将在第 5 章中介绍的方法 `FullScreenToggler.monitor()` 用于允许用户在窗口模式和全屏模式之间进行切换。类

ifecycleWindowListener 中静态的便利方法 `launchFrameAsDesktopApp` 显示出该帧，并且当用户激活窗口、使窗口无效或关闭窗口的时候，对 `MultiApplet` 实例调用适当的生命周期方法。

```
public MultiApplet (
    String    appletInfo,
    Pair [ ] appletPairs,
    String    newsName,
    String    newsHTML,
    String    newsPage )
    //////////////////////////////////////
{
    NullPointerException.check ( this.appletInfo = appletInfo );

    NullPointerException.check ( this.appletPairs = appletPairs );

    NullPointerException.check ( this.newsName = newsName );

    this.newsHTML = newsHTML;

    this.newsPage = newsPage;
}
```

构造函数只是简单地保存对构造函数参数的引用，留作在后面的 `init()` 方法中进行使用。

```
public String getAppletInfo ( )
    //////////////////////////////////////
{
    return appletInfo;
}

public void init ( )
    //////////////////////////////////////
{
    Container contentPane = getContentPane ( );

    contentPane.setLayout ( new BorderLayout ( ) );

    jTabbedPane = new JTabbedPane (
        JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT );

    contentPane.add ( jTabbedPane, BorderLayout.CENTER );
}
```

`init()` 方法使用 `JtabbedPane` 填充整个帧。标签沿着窗口顶部排列，并且当有太多标签填满整个窗口宽的时候，它还可以从左到右滚动。通过使用滚动标签，可以获得很多标签，每一个标签对应一个游戏，而不需要耗尽所有的屏幕资源。

```
jTabbedPane.add (
    new MultiAppletNews ( newsHTML, newsPage, this ), newsName );
```

在第一个标签上，安放了一个称为 `MultiAppletNews` 的特殊组件。我对这个标签的处理方法与其他标签是有区别的，因为它从我的 Web 站点上下载一个 Web 页面。我不想让它在

用户单击其他标签时被销毁，因为如果用户后来又要回到这个页面时，它还需要再次下载这个 Web 页面。因此，正如下面的代码所给出的一样，代码对这个标签的处理与其他标签的处理是不同的。

```
for ( int i = 0; i < appletPairs.length; i++ )
{
    jTabbedPane.add ( new JPanel ( ), appletPairs [ i ].name );
}
```

然后，再创建其他标签，并标记上游戏名称的简写。

```
jTabbedPane.addChangeListener (
    new ChangeListener ( )
    {
        public void stateChanged ( ChangeEvent changeEvent )
        {
            handleStateChange ( );
        }
    } );

multiAppletStub = new MultiAppletStub ( this );
}
```

向JTabbedPane中添加一个ChangeListener，这样，就可以知道用户单击不同的标签的时间。在初始化方法的最后创建了一个MultiAppletStub实例。

```
public void start ( )
{
    //////////////////////////////////////
    multiAppletStub.setActive ( true );

    try
    {
        if ( appletComponent instanceof Applet )
        {
            ( ( Applet ) appletComponent ).start ( );
        }
        else if ( appletComponent instanceof Lifecycle )
        {
            ( ( Lifecycle ) appletComponent ).start ( );
        }
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }

    isStarted = true;
}
```

如果它能确定它是 Applet 的一个实例或是 Lifecycle 的一个实例，MultiApplet 的 start()方

法就会委托给当前 `appletComponent` 的 `start()` 方法。注意，对包含在 `MultiApplet` 内的游戏来说，它就必须扩展 `Component`，但是它不需要扩展 `Applet` 或实现 `Lifecycle`。这样的实例可能就算是一个这样的游戏：这种游戏动画的更新是靠用户输入事件(例如单击鼠标)驱动的，而不是由一个连续运行的动画线程驱动的。

```
public void stop ( )
///////////////////////////////////////////////////
{
    multiAppletStub.setActive ( false );

    try
    {
        if ( appletComponent instanceof Applet )
        {
            ( ( Applet ) appletComponent ).stop ( );
        }
        else if ( appletComponent instanceof Lifecycle )
        {
            ( ( Lifecycle ) appletComponent ).stop ( );
        }
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }

    isStarted = false;
}

public synchronized void destroy ( )
///////////////////////////////////////////////////
{
    try
    {
        if ( appletComponent instanceof Applet )
        {
            ( ( Applet ) appletComponent ).destroy ( );
        }
        else if ( appletComponent instanceof Lifecycle )
        {
            ( ( Lifecycle ) appletComponent ).destroy ( );
        }
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }
}
```

`stop()` 方法和 `destroy()` 方法类似。`start()` 方法和 `stop()` 方法改变着 `MultiAppletStub` 的活动状态。这些方法的调用通常与窗口事件(例如窗口激活或非激活、最大化最小化窗口或关闭窗口)相对应。

```
private void handleStateChange ( )
///////////////////////////////////////////////////
{
    if ( isStarted )
    {
        stop ( );
    }
}
```

如果用户单击了一个不同标签，并且当前选中的游戏正在运行，handleStateChange()方法就会将它停止。

```
if ( index > 0 )
{
    jTabbedPane.setComponentAt ( index, new JPanel ( ) );

    destroy ( );

    appletComponent = null;

    System.gc ( );
}

index = jTabbedPane.getSelectedIndex ( );
```

index 为 0 的位置表示正在显示包含 MultiAppletNews 的标签面板。只要当前 index 的位置不为 0，当前的 appletComponent 就会被销毁，所有指向它的引用也都会被删除。系统内存的垃圾回收就会在这一点强制执行垃圾回收功能，以减少它会自动运行并中断新选择的动画的这种概率。index 的位置被重新调整到指向新选择标签面板的位置。

```
if ( index > 0 )
{
    try
    {
        appletComponent = ( Component ) Class.forName (
            appletPairs [ index - 1 ].value ).newInstance ( );
    }
```

如果新的 index 位置不是 0，那么就会使用动态链接的方法将游戏加载到内存中。

```
if ( appletComponent instanceof Applet )
{
    ( ( Applet ) appletComponent ).setStub ( multiAppletStub );
}

if ( appletComponent instanceof JComponent )
{
    FullScreenToggler.monitor ( ( JComponent ) appletComponent );
}

jTabbedPane.setComponentAt ( index, appletComponent );
```

如果 appletComponent 是 Applet 的一个实例，就会设置 AppletStub。如果它是 JComponent 的一个实例，我们就可以监控它进入全屏模式的键盘事件。所有动态加载的类都至少要扩展 Component，这样它们才能被加载到 JTabbedPane 中去。


```

try
{
    if ( appletComponent instanceof Applet )
    {
        ( ( Applet ) appletComponent ).init ( );
    }
    else if ( appletComponent instanceof Lifecycle )
    {
        ( ( Lifecycle ) appletComponent ).init ( );
    }
}
catch ( Exception ex )
{
    ex.printStackTrace ( );
}

start ( );
}
catch ( Exception ex )
{
    ex.printStackTrace ( );
}
}
}

```

在新创建的 `appletComponent` 实例的 `init()` 方法的后面，紧接着调用 `MultiApplet` 的 `start()` 方法。注意，`MultiApplet` 的 `init()` 方法初始化 `MultiApplet`，但是它的 `start()` 方法初始化了 `appletComponent`。

2.4.6 CroftSoftCollection

`CroftSoftCollection` 是一个示范程序，当在 Ant 构建文件中运行默认目标的时候，就会编译并启动这个程序。通过修改常量的方法，可以将这个程序作为一个模板代码来在 `MultiApplet` 中发布自己的游戏集。

```

package com.croftsoft.apps.collection;

import java.awt.Dimension;
import java.net.URL;

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.gui.multi.MultiApplet;
import com.croftsoft.core.lang.Pair;

public final class CroftSoftCollection
    extends MultiApplet
{
    //////////////////////////////////////
    //////////////////////////////////////
    {
        [...]
    }
}

```

CroftSoftCollection 扩展了 MultiApplet, 因此它可以在 applet 容器中运行。

```
private static final Pair [ ] APPLET_PAIRS = {
    new Pair (
        "BackpropXOR",
        "com.croftsoft.apps.backpropxor.BackpropXor" ),
    new Pair (
        "Clock",
        "com.croftsoft.apps.clock.DigitalClock" ),

    [...]
    new Pair (
        "Zombie",
        "com.croftsoft.apps.zombie.Zombie" ) };
```

它定义了 APPLET_PAIRS 数组, 并将这个数组作为 MultiApplet 的一个参数传递过去。对数组的长度没有预定义的限制。无论什么时候创建新游戏, 通常都会将它按照字母表的顺序插入到这个数组中。前面的代码只给出了其中的 3 个数组元素。

```
private static final String NEWS_NAME
    = MultiApplet.DEFAULT_NEWS_NAME;

private static final String NEWS_HTML
    = "<html><body><pre>" + APPLET_INFO + "</pre></body></html>";

private static final String NEWS_PAGE
    = "http://www.croftsoft.com/portfolio/collection/news/";
```

如果您自己创建 MultiApplet 子类部署游戏, 也许想改变这些 MultiAppletNews 常量。也可能希望能够从 JAR 内部的源文件中下载 NEWS_HTML 的文本, 而不是像我在这里这样将它定义为经过编译以后的常量。

```
private static final String FRAME_TITLE = TITLE;

private static final String FRAME_ICON_FILENAME
    = CroftSoftConstants.FRAME_ICON_FILENAME;

private static final Dimension FRAME_SIZE = null;

private static final String SHUTDOWN_CONFIRMATION_PROMPT
    = "Close " + TITLE + "?";
```

还可以在自己的子类中定制帧变量。也可能希望首先备份 CroftSoftCollection.java, 更改包和类的名称, 并用自己的一个游戏替换其中一个游戏。

```
public static void main ( String [ ] args )
    throws Exception
    //////////////////////////////////////
    {
        System.out.println ( APPLET_INFO );
```



```

MultiApplet.launch (
    APPLET_INFO,
    APPLET_PAIRS,
    NEWS_NAME,
    NEWS_HTML,
    NEWS_PAGE,
    FRAME_TITLE,
    FRAME_ICON_FILENAME,
    CroftSoftCollection.class.getClassLoader ( ),
    FRAME_SIZE,
    SHUTDOWN_CONFIRMATION_PROMPT );
}

```

注意，main()方法并没有创建 CroftSoftCollection 的实例。它只是简单地将在那个类中定义的静态常量传递给 MultiApplet 的静态方法 launch()。

```

public CroftSoftCollection ( )
///////////////////////////////////////////////////
{
    super (
        APPLET_INFO,
        APPLET_PAIRS,
        NEWS_NAME,
        NEWS_HTML,
        NEWS_PAGE );
}

```

但是，当 CroftSoftCollection 以 applet 的形式使用的时候，必须提供一个无参数的构造函数，这个函数将这些常量传递给超类 MultiApplet 的构造函数。

```

<javac srcdir="${srcdir}" destdir="jar">
  <include name="com/croftsoft/apps/collection/CroftSoftCollection.java"/>
  <include name="com/croftsoft/apps/backpropxor/BackpropXor.java"/>
  <include name="com/croftsoft/apps/clock/DigitalClock.java"/>
  [...]
  <include name="com/croftsoft/apps/zombie/Zombie.java"/>
</javac>
<jar
  jarfile="collection.jar"
  basedir="jar"
  manifest="bld/apps/collection/manifest.txt"/>

```

因为这些游戏 applet 都是动态链接的，所以，必须谨慎地确定它们都在构建中。前面的代码是 build.xml 文件的一部分。注意，如果类是动态链接的，就只需要第一个 include 标记。

2.5 小结

本章主要介绍了 3 种标准的 Java 游戏部署框架：基于浏览器的 applet、可执行的 JAR 文件和 Java Web Start。深入讨论了每种部署框架的优缺点，以及包括安全性在内的各种不同特性。这一章还介绍了创建自己游戏的多种编程的技术，包括动画线程管理、使用定制接口和动态链

接来分离可选包、外部浏览器通过 JNLP 访问、将帧用作单独的容器、使用 Swing 显示 Web 页面。本章最后对 MultiApplet 的源代码进行了详细的研究，这个 MultiApplet 是一个可以直接用来部署游戏的可重用框架。

2.6 参考文献

Schmidt, Rene W. *Java Network Launching Protocol & API Specification*. <http://java.sun.com/products/javawebstart/download-spec.html>.

Sun Microsystems. *Java Plug-in 1.4.2 Developer Guide*. <http://java.sun.com/j2se/1.4.2/docs/guide/plugin/>.

Sun Microsystems. *Java Web Start 1.4.2 Developer Guide*. <http://java.sun.com/j2se/1.4.2/docs/guide/jws/developersguide/contents.html>.

Sun Microsystems. "Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?" <http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html>.

Topley, Kim. "JEditorPane and the Swing HTML Package." Chapter 4 in *Core Swing Advanced Programming*. Upper Saddle River, NJ: Prentice-Hall PTR, 2000.

第 3 章 Swing 动画

不要浪费时间，因为它是生命之构成。

—— 本杰明·富兰克林

本章将详细地介绍在基于 Swing 的可重用动画库中起到支柱作用的 3 个接口和 1 个类。还将附带介绍每一个接口的一个或多个实现。掌握这些内核类的操作过程是理解折衷实现的关键。在开始创建和集成自己的动画类，创建一个独具特色的游戏时，理解这些类的工作过程是非常有用的。

- ComponentAnimator
- RepaintCollector
- LoopGovernor
- AnimatedComponent

这些类协同工作，在包含有 3 个主要阶段的动画循环中产生连续的动画帧。第 1 个阶段也叫更新阶段，在这一阶段中，ComponentAnimator 更新精灵的位置，并产生重绘请求。这些重绘请求由 RepaintCollector 进行收集和合并。在第 2 个阶段，也就是绘制阶段，向 RepaintCollector 查询组件中那些需要重绘的区域，然后由 AnimatedComponent 对这些需要重绘的区域进行重绘。在第 3 个阶段中，也就是延迟阶段，使用 LoopGovernor 延迟正在运行动画循环的线程，让该动画循环的长度能够满足想要帧速率的长度要求。AnimatedComponent 是一个 Swing 组件，该组件通过提供发生绘画操作的区域、调用其他类的动画循环，以及让动画可以集成到框架的生命周期方法，从而将绘制的区域组合在一起。

3.1 ComponentAnimator

我们的第一个内核动画类是 ComponentAnimator。接口 ComponentAnimator 的实现负责更新精灵的位置、标识出组件的哪些区域需要进行最终的重绘并在组件的表面上重绘这些区域。ComponentAnimator 的实现包含了游戏特定的代码：

```
package com.croftsoft.core.animation;

public interface ComponentAnimator
    extends ComponentUpdater, ComponentPainter
```

ComponentAnimator 接口扩展了 ComponentUpdater 和 ComponentPainter 这两个接口。这个接口本身并没有定义其他的方法。

```
public void update ( JComponent component );
```

接口 `ComponentUpdater` 就只定义了一个方法——`update(JComponent)`——这个方法用于更新精灵的位置，以及请求组件的重绘——只要需要的话。`update(JComponent)`通过调用 `JComponent` 的重绘方法，以及包含从 `JComponent` 超类 `Component` 继承的那些重绘方法，产生重绘请求。注意 `JComponent` 的特殊子类 `AnimatedComponent` 重写了它超类 `JComponent` 的 `repaint()`方法，以提高动画的性能。`ComponentUpdater` 可以和 `JComponent` 的任何子类一起使用，而不只是 `AnimatedComponent`。

```
public void paint (
    JComponent component,
    Graphics2D graphics );
```

`ComponentPainter` 接口定义一个对象，这个对象知道怎样和在哪里绘制图像 `JComponent`。可以将上面给出的 `paint()`方法的签名与下面给出的 `paintIcon()`方法的签名(在 `Swing` 接口 `Icon` 中定义的)进行对比。

```
public void paintIcon (
    Component c,
    Graphics g,
    int x,
    int y );
```

通过比较可以发现 `Swing Icon` 知道怎样绘制组件，但是不知道应该在哪里进行绘制。这个“在哪里进行绘制”的位置必须由坐标 `x` 和 `y` 给出。然而，`ComponentPainter` 被假定为已经知道了在哪里绘制组件，并且常常会将这些坐标封装为内部的状态(如果需要的话)。

还有两个细微的区别：`ComponentPainter` 接口在它的方法参数中，使用的是 `JComponent` 而不是其超类 `Component`，使用的是 `Graphics2D` 而不是其超类 `Graphics`。这说明 `ComponentPainter` 的实现不需要将 `Graphics` 引用转换为 `Graphics2D` 引用，从而获取 `Graphics2D` 对象的扩展功能。

3.1.1 更新和绘制阶段

通常在两个阶段中，可以通过动画循环访问 `ComponentAnimator`。首先，在更新阶段中，调用它的 `update()`方法更新精灵的位置，并产生重绘请求。其次，在绘制阶段，调用它的 `paint()`方法，在精灵的新位置上重新绘制该精灵的组件。

将这两个操作分隔开的理由有两个。第一个理由是这样能使重绘组件的工作更加简单，而不需要去更新精灵的位置。这通常都是必需的，例如动画暂停的时候，因为组件可能被其他窗口暂时性覆盖一下，而后又解除覆盖，这时就需要重绘。第二个理由是，这样能使在发生绘制事件之前，一次执行所有精灵位置的更新和绘制请求更加有意义。这是因为精灵是同时移动的，您不希望在它的新位置上绘制其他被遮叠精灵的同时，在精灵的原位置上还要重绘一部分它的图像(覆盖住的精灵就不需要再进行重绘了)。这样能使收集和与重绘请求的结合更加简单，因为在真正发生任何绘制操作之前，这些请求可能会被合并在一起。

3.1.2 精灵的定义

记住，我在使用精灵(sprite)这个术语的时候，实际上使用的是一个非常松散的定义，在这种定义中，精灵可以是能够更新状态、绘制组件的任何东西，或者可以被调用去更新状态、绘

制组件的任何东西。本书中，在我使用这个词而又没有进行具体说明它是什么的时候，一般指的就是 `ComponentUpdater`、`ComponentPainter` 或 `ComponentAnimator` 接口的一个实现。

这与精灵传统的定义是有区别的，或与有些书中给出的参与者(actor)的概念也是有区别的，参与者好像是既可以更新状态，又能够绘制像 `ComponentAnimator` 这样(但是带有一些附加限制)的组件的一个对象。首先，术语精灵通常指的是场景中的一个图标参与者，例如字符，与背景或装饰相对。基于这种定义，一个太空船就可以是一个精灵，但是在背景上描述星星滑动区域的对象就不能称为一个精灵。其次，它常常会定义一些与运动相关的属性和处理运动的方法。下一章中还要介绍 `ComponentAnimator` 的一个子接口，这个子接口的名称为 `Sprite`。在这个接口中，定义了空间坐标、速度和标题等一些属性，并提供了一些方法，来判定在虚拟空间中对象是否与其他对象发生了碰撞。

但是，这一章并没有使用这种具体的规定。精灵可以是组件的背景颜色、场景中的非运行目标、雾化的效果、显示当前得分的文本或作为单个合成对象的一群较小对象。它可以是从动画循环中调用更新状态或绘制组件的任何事物。

3.1.3 ExampleAnimator

```
package com.croftsoft.ajgp.anim;

import java.awt.Graphics2D;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.lang.NullArgumentException;

public final class ExampleAnimator
    implements ComponentAnimator
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    private final String text;

    private final int    deltaX;

    private final int    deltaY;

    //

    private int x;

    private int y;

    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    public ExampleAnimator (
        String text,
        int    deltaX,
        int    deltaY )
```

```

////////////////////////////////////
{
    NullPointerException.check ( this.text = text );

    this.deltaX = deltaX;

    this.deltaY = deltaY;
}

////////////////////////////////////
////////////////////////////////////

public void update ( JComponent component )
////////////////////////////////////
{

    x += deltaX;

    y += deltaY;

    int componentWidth = component.getWidth ( );

    int componentHeight = component.getHeight ( );

    if ( x > componentWidth )
    {
        x = 0;
    }
    else if ( x < 0 )
    {
        x = componentWidth;
    }

    if ( y > componentHeight )
    {
        y = 0;
    }
    else if ( y < 0 )
    {
        y = componentHeight;
    }

    component.repaint ( );
}

public void paint (
    JComponent component,
    Graphics2D graphics )
////////////////////////////////////
{
    graphics.setColor ( component.getForeground ( ) );

    graphics.drawString ( text, x, y );
}

```


包 `com.croftsoft.ajgp.anim` 中的 `ExampleAnimator` 是 `ComponentAnimator` 的一个简单实现，它将文本滑过组件。在 `update()` 方法中，`x` 和 `y` 坐标分别使用 `deltaX` 和 `deltaY` 进行滑动，并在每一个新位置均产生一个动画。如果文本滑出了组件的边界，它就会从另一边界滚出。`update()` 方法的最后一个动作是请求重绘整个组件，以显示更新过的位置。在后面处理重绘请求的时候，`paint()` 方法只是简单地在新位置上绘制出文本。

3.2 RepaintCollector

可以在编写 Java 游戏程序过程中采用的一件最有意义的性能优化工作就是，将在动画的每一个新帧中要绘制像素的数目降低到最小，这是我多年来得出的经验。有时，可以简单地重绘整个屏幕，仍然可以达到想要的帧速率要求。例如 `ExampleAnimator` 就是在每次调用它的 `update()` 方法的时候，请求对整个组件进行重绘。但是，在较慢的计算机上，或者在没有图像加速功能的计算平台上，动画的速度可能就变得很慢，以至于游戏就不能玩了。下面几个小节将介绍一种新的机制，使用这种机制能确保您的游戏能够在多种平台上，并以玩家可以接受的帧速率运行。

3.2.1 Swing 串行化

Swing GUI 库的代码已经在性能方面进行了优化，这就意味着几乎没有同步的方法可以阻止方法被不同的线程同时调用。换句话说，也就是大多数 Swing 方法都不是线程安全的，都应该串行地执行，例如一次只能执行一个方法。如果冒险同时调用了这些方法，有时就会看到一些滑稽的效果，例如组件在响应多个并发的外部事件时，可能会不能正常地在屏幕上重绘它自身。

但是在绝大部分时间里，这都不会成为问题，因为 Swing 的方法(例如 `repaint()` 方法)实际上不会立即重绘屏幕，而是简单地在一个队列里将请求排队，然后再使用另一个特殊的线程串行地执行。这个特殊的线程被称为事件分派线程(event dispatch thread)，它持续地监控 AWT `EventQueue`，使操作能够串行地执行。

`repaint()` 方法相当灵活，如果多个重绘请求在 AWT `EventQueue` 队列中排队的速度比事件分派线程能够处理的速度要快，那么这些重绘请求就能够进行合并，例如合并到单个请求中，以提高效率。如果多个请求的重绘区域并不完全相同，新合并的请求将会包含一个新矩形区域，在这个新矩形区域中包含那两个原始请求区域的联合区域。

3.2.2 动画的问题

在 Swing 中，当动画的速度很高时，在效率方面的某些努力将会导致相反的效果。一般而言，它单独处理每一个请求所花的时间比处理这些请求合并成的新请求所花费的时间还要少。例如，假设有一个精灵在屏幕(屏幕的背景是不变的)的左上角漫游。仅通过在一个非常小的区域上重绘覆盖精灵原位置的背景，并在精灵的新位置上重新绘制另一个精灵，重绘请求就可以

快速地得到处理。如果精灵的长和宽都是 32 位像素(32×32),这其中涉及到需要重绘的像素数目为 $2 \times 32 \times 32 = 2048$ (像素)。

通常可以绘制更少的像素,因为新精灵的区域可能会与精灵的原区域有重叠。例如,如果精灵的最大速度和帧速率正好使精灵每帧最多只能移动一个像素的话,就可以仅重绘一个 32×32 大小的像素区域。也就是 1089 个像素,正好是 2048 个像素(如果我们分别重绘两个精灵的话,需要重绘的像素就是 2048)的一半。在这种情况下,将两个重绘请求——一个用来重绘精灵的原位置,另一个用来重绘精灵的新位置——合并为一个重绘请求对我们会有帮助。在即时(Just-In-Time, JIT)的编译器、千兆赫兹的计算机和图像加速设备出现以前,这是我在 Java 早期为了达到加快动画效果而常常使用的一个技巧。

但是,我们现在假定有两个精灵漫游在背景不变的场景中,一个向右上角移动,而另一个向左下角移动。如果重绘这两个精灵的请求是单独处理的,那么只有 $2 \times 33 \times 33 = 2178$ 个像素需要重绘。但是,如果这两个请求被合并到一个请求中,那就是完全不同的另一件事情了。假定这两个精灵在水平和垂直方向的距离都是 300 个像素。无论哪一个合并的重绘请求,都将至少要重绘这 300×300 像素的一个矩形区域,这就是 90000 个额外需要重绘的像素。如果当这两个精灵相向移动的时候,动画则会慢慢加速;但是在反向移动的时候,动画必然会越来越慢。

可以使用 CraftSoft Collection 中的 Sprite 程序来观察这个现象,如图 3-1 所示。Sprite 程序启动的时候,目标帧速率被设为默认的最大值,度量的单位是帧每秒(fps)。由于现在的大部分机器不能那么快地重绘组件,实际达到的帧速率将会减少。在 Options 菜单项下面,选择 Only sprite regions 菜单命令。帧速率应该立刻会明显地提高,因为在每一帧中,只有精灵区域而不是整个组件区域都被重绘。您应该可以清楚地区分这些重绘的区域,因为背景墙砖的图案会在它们边界以内移动,而不会在它们的边界以外移动。对这种现象观察一段时间以后,选择选项 Use Swing RepaintManager。两个精灵的重绘区域就会合并为一个重绘区域,当这两个精灵反向运动的时候,这个重绘区域就会加大,而当这两个精灵相向运动的时候,这个重绘区域就会逐渐减少。帧速率将会与这个区域的大小成反比。



图 3-1 合并重绘请求

动画的默认 Swing 重绘行为的另外一个问题是，有些重绘可能会在您想让它们重绘之前就重绘了。假定有两个精灵，它们左右排列，一起运动。在这种情况下，您可能希望首先更新第一个精灵的位置，然后再更新第二个精灵的位置，最后再同时重绘这两个精灵，这样在某一瞬间就不会有一个精灵会覆盖另一个精灵。默认 Swing 的重绘行为是分别重绘每一个精灵，还是同时重绘这两个精灵，取决于重绘请求的时间以及请求合并与否。

3.2.3 RepaintCollector

类 `AnimatedComponent` (将在本章的最后详细地介绍这个类) 可以通过将所有的重绘请求都委托给包 `com.croftsoft.core.animation` 中接口 `RepaintCollector` 的实例，重写重绘操作默认的行为。`RepaintCollector` 负责收集在更新阶段产生的重绘请求。这使您可以提供自己想要怎样合并多个请求的策略，也可以提供用来控制什么时候处理它们的策略。

```
public int getCount ( );

public Rectangle [ ] getRepaintRegions ( );
```

接口 `RepaintCollector` 的 `getCount()` 方法返回在当前动画循环迭代过程中产生的重绘请求的数目。而请求重绘的区域是由 `getRepaintRegions()` 方法返回的。返回的 `Rectangle` 数组的长度可能会比请求的数目还要长一些；在这种情况下，只有第一个 `count` 的位置才会包含有效的数据。

```
public void repaint ( );

public void repaint (
    int x,
    int y,
    int width,
    int height );
```

`AnimatedComponent` 使用两个方法将重绘请求委托给 `RepaintCollector`。第一个方法是请求重绘整个组件，第二个方法是产生一个重绘组件某一个矩形区域的请求。绘制整个组件是很有用的，尤其是当整个背景都在移动的时候。

```
public void reset ( );
```

重新绘制组件以后，就会在每一个动画循环的末尾调用 `reset ()` 方法。调用这个方法就是要将请求计数重新设为 0，为下一个循环做好准备。

3.2.4 SimpleRepaintCollector

```
public class SimpleRepaintCollector
    implements RepaintCollector
```

`SimpleRepaintCollector` 是 `RepaintCollector` 接口的一个非常简单的实现。它在 `com.croftsoft.core.animation.collector` 包中。子包 `collector` 中还包含一些其他 `RepaintCollector` 的具体实现，例如 `BooleanRepaintCollector`、`CoalescingRepaintCollector`、`NullRepaintCollector` 和 `SwingRepaintCollector`。现在，我的动画类就是按照子包(与接口具有相同名称的子包)内部接口的具体实现进行分组的。

```

private int      count;

private Rectangle [ ] repaintRegions;

public SimpleRepaintCollector ( )
///////////////////////////////////////////////////
{
    repaintRegions = new Rectangle [ 0 ];
}

public int getCount ( ) { return count; }

public Rectangle [ ] getRepaintRegions ( )
///////////////////////////////////////////////////
{
    return repaintRegions;
}

```

SimpleRepaintCollector 对重绘请求的响应就是递增请求计数和存储请求区域。当调用它的 **accessor** 方法时, **SimpleRepaintCollector** 只是简单地返回特定的存储数据。

```

public void repaint (
    int x,
    int y,
    int width,
    int height )
///////////////////////////////////////////////////
{
    if ( count == repaintRegions.length )
    {
        repaintRegions = ( Rectangle [ ] ) ArrayList.append (
            repaintRegions, new Rectangle ( x, y, width, height ) );
    }
    else
    {
        repaintRegions [ count ].setBounds ( x, y, width, height );
    }

    count++;
}

```

重绘一个区域的请求只是简单地追加到 **Rectangle** 数组 **repaintRegions** 中, 也同时递增 **count**。如果数组不够长, 就会用一个新数组(这个新数组使用了 **com.croftsoft.core.util** 包的 **ArrayList.append()**方法)将其替换。如果数组够长, 数组中原来的 **Rectangle** 实例就会重新用来保存新的数据。假定数组的长度最终达到某一个最大的值, 也许与游戏中精灵的数目相等, 这个技术保证了不必为每一个动画循环创建一个新的 **Rectangle** 实例。

这个方法是异步的, 因为一般都有这样的假定: 在更新阶段产生的重绘请求都是要串行地通过事件分派线程执行的。如果不这样的话, 这个方法就会被多个线程同步调用, 数据就会变得混乱。


```

public void repaint ( )
///////////////////////////////////////////////////
{
    repaint ( 0, 0, Integer.MAX_VALUE, Integer.MAX_VALUE );
}

```

`ArrayLib.append()`方法处理重绘整个组件的请求。这个实现只是简单地委托给前面使用了最大尺寸的 `repaint()`方法。

```

public void reset ( )
///////////////////////////////////////////////////
{
    count = 0;
}

```

`reset` 方法简单地将 `count` 设为 0。`Rectangle` 数组 `repaintRegions` 没有被清空，因为在下一个循环中，原数据将会直接被新数据覆盖。

3.2.5 BooleanRepaintCollector

`BooleanRepaintCollector` 是 `RepaintCollector` 的另一个简单实现，在这个实现中，无论对一个或者是多个重绘请求，它都是简单地重绘整个组件，而不管请求重绘区域的大小。如果在动画循环过程中没有重绘请求，就不会进行任何重绘。考虑该 `RepaintCollector` 实现的另外一种思路是，将其作为一个将多个请求合并为一个请求的 `all-or-nothing` 策略。

`BooleanRepaintCollector` 通常能够满足多个动画任务的需求。原因很简单，如果最少有一个重绘请求，这个实现就会在每一帧重绘整个组件，所以在更新精灵位置的时候，`ComponentAnimator` 的实现就不需要计算重绘区域。如果场景的背景一直在到处移动，而您知道整个组件无论如何都必须要重绘的时候，这就显得非常有意义。

```

private static final Rectangle [ ] REPAINT_REGIONS
    = new Rectangle [ ] {
        new Rectangle ( Integer.MAX_VALUE, Integer.MAX_VALUE ) };

//

private boolean doRepaint;

```

`Rectangle` 数组常量 `REPAINT_REGIONS` 就包含一个最大的矩形区域。布尔变量 `doRepaint` 是一个标志，它用来表示在当前动画循环迭代的过程中是否有重绘请求。

```

public int getCount ( )
///////////////////////////////////////////////////
{
    return doRepaint ? 1 : 0;
}

public Rectangle [ ] getRepaintRegions ( )
///////////////////////////////////////////////////
{
    return REPAINT_REGIONS;
}

```

`getCount()`方法一般都会在当前动画循环迭代的过程中，简单地返回重绘请求值(如果有任意数目的重绘请求的话)。如果 `getCount()` 返回值 0，`getRepaintRegions()` 方法返回的数组将不被使用，因为只有它的第一个 0 元素才被认为包含有效数据。

```
public void repaint (
    int x,
    int y,
    int width,
    int height )
///////////////////////////////////////////////////
{
    doRepaint = true;
}

public void repaint ( )
///////////////////////////////////////////////////
{
    doRepaint = true;
}
```

这两个方法设置了 `doRepaint` 标志。注意，并没有使用第一个方法中的那个区域参数。如果精灵没有移动，这两个方法也就都得不到调用。在这种情况下，`doRepaint` 标志的值为 `false`。

```
public void reset ( )
///////////////////////////////////////////////////
{
    doRepaint = false;
}
```

在每一个动画循环的结尾处，`doRepaint` 标志都被设为 `false`。

3.2.6 CoalescingRepaintCollector

当有重绘组件一小块区域的请求(这种请求的中间可能还夹着重绘整个组件的请求)时，使用 `SimpleRepaintCollector` 可能就不太合适了。在这种情况下，当需要一次或多次重绘整个组件时，您就会在完成重绘小块区域以后，立即需要在同一帧中再次重绘整个组件来覆盖该区域。在这种情况下，使用 `BooleanRepaintCollector` 能够更好地将所有这些请求合并为一个请求。

但是，当有仅需要重绘组件的几个小区域的请求时，使用 `BooleanRepaintCollector` 可能就不太合适了。在这种情况下，就没有必要重绘整个组件，因为这样会使动画的速率降低。我们这时所需要的是一个新机制，这种新机制最好能够足够灵活，当只有几个小区域需要重绘的时候，它能像 `SimpleRepaintCollector` 一样，而当至少有一个请求要重绘整个组件的时候，它就能够像 `BooleanRepaintCollector` 一样。

在每一个动画循环开始的时候，`CoalescingRepaintCollector` 就分别存储重绘请求。但是，如果它收到重绘整个组件的请求时，它就会将多个重绘请求合并为一个。在每一个动画循环中，如果预先不知道 `ComponentAnimator` 是否会产生重绘整个组件的请求，`CoalescingRepaintCollector` 就是一个很好用的实现方法。


```

package com.croftsoft.core.animation.collector;

import java.awt.Rectangle;
import java.awt.geom.Rectangle2D;

import com.croftsoft.core.animation.RepaintCollector;
import com.croftsoft.core.util.ArrayLib;

public class CoalescingRepaintCollector
    implements RepaintCollector
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final Rectangle [ ] ALL_REGIONS = new Rectangle [ ] {
        new Rectangle ( Integer.MAX_VALUE, Integer.MAX_VALUE ) };

    //

    private int                count;

    private boolean            repaintAll;

    private Rectangle [ ]      repaintRegions;

    //////////////////////////////////////
    //////////////////////////////////////

    public CoalescingRepaintCollector ( )
    //////////////////////////////////////
    {
        repaintRegions = new Rectangle [ 0 ];
    }

```

布尔变量 `paintAll` 是一个标志，用来标识在该动画循环迭代的过程中，是否应该重绘整个组件。正如您可能看到的一样，`CoalescingRepaintCollector` 包含有与 `SimpleRepaintCollector` 和 `BooleanRepaintCollector` 相同的大部分实例变量。

```

    public int getCount ( )
    //////////////////////////////////////
    {
        if ( repaintAll )
        {
            return 1;
        }

        boolean hasIntersections = true;

        while ( hasIntersections )
        {
            hasIntersections = false;

```

```

iLoop:

for ( int i = 0; i < count - 1; i++ )
{
    Rectangle iRectangle = repaintRegions [ i ];

    for ( int j = i + 1; j < count; j++ )
    {
        Rectangle jRectangle = repaintRegions [ j ];

        if ( iRectangle.intersects ( jRectangle ) )
        {
            hasIntersections = true;

            Rectangle2D.union ( iRectangle, jRectangle, iRectangle );

            repaintRegions [ j ] = repaintRegions [ count - 1 ];

            repaintRegions [ count - 1 ] = jRectangle;

            count--;

            break iLoop;
        }
    }
}

return count;
}

public Rectangle [ ] getRepaintRegions ( )
///////////////////////////////////////////////////
{
    return repaintAll ? ALL_REGIONS : repaintRegions;
}

```

如果设置了 `repaintAll` 标志，`CoalescingRepaintCollector` 的 `accessor` 方法的返回值就与 `BooleanRepaintCollector` 的返回值相同。如果 `repaintAll` 标志被置为 `false`，其 `accessor` 方法的返回值就与 `SimpleRepaintCollector` 的返回值相同，不同的就是在调用 `getCount()` 方法的时候，重叠的重绘区域会被合并在一起。但是，标准的 `Swing RepaintManager` 会将屏幕上所有的重绘区域——不管它们是分隔多远——都合并到一个较大的单独重绘区域中，而 `CoalescingRepaintCollector` 仅仅合并那些重叠在一起的区域。

```

public void repaint (
    int x,
    int y,
    int width,
    int height )
///////////////////////////////////////////////////

```



```

{
    if ( repaintAll )
    {
        return;
    }

    if ( count == repaintRegions.length )
    {
        repaintRegions = ( Rectangle [ ] ) ArrayList.append (
            repaintRegions, new Rectangle ( x, y, width, height ) );
    }
    else
    {
        repaintRegions [ count ].setBounds ( x, y, width, height );
    }
    count++;
}

```

设置了 `repaintAll` 标志以后，就会简单地忽略产生的所有请求。

```

public void repaint ( )
///////////////////////////////////////////////////////////////////
{
    repaintAll = true;
}

public void reset ( )
///////////////////////////////////////////////////////////////////
{
    count = 0;

    repaintAll = false;
}

```

在动画循环迭代过程中，首次请求重绘整个组件时，会设置 `repaintAll` 标志，在迭代结束时，会重置该标志。

3.2.7 其他实现

如果您觉得确实能够提高动画的性能，就会很愿意创建自己的 `RepaintCollector` 接口实现。例如，我做了一个这样的实现：在达到某一个最大数目以后，就将多个小的重绘请求合并到一个请求中。当有很多精灵在整个屏幕上涌动的时候，这种实现可能会很有效。由于只有您自己才最清楚游戏的具体细节问题，所以能够为 `RepaintCollector` 定制一个实现，这种定制的实现可能比其他任何普通的实现都更加有效。

3.3 LoopGovernor

在动画阶段，速率就是一切。如果帧速率太快，产生的新帧可能会比人眼能够分辨的速率

还要快，或者比显示器能够显示的速率还要快，甚至比精灵能够移动的速率还要快，这无形中就浪费了不必要的时间。如果帧速率太低或帧之间的时间间隔不一致，动画就会发生抖动。

如图 3-2 所示，可以使用 Sprite 示例，观察动画的平滑性。在选项中，关闭除第一个精灵以外的所有精灵的绘制，同时可能还需要关闭双缓冲。将目标帧速率调整到某些合理的值，例如 30fps。由于没有绘制背景，应该观察到在新位置上绘制第一个精灵的时候，原来绘制的那个精灵还残留在屏幕上。

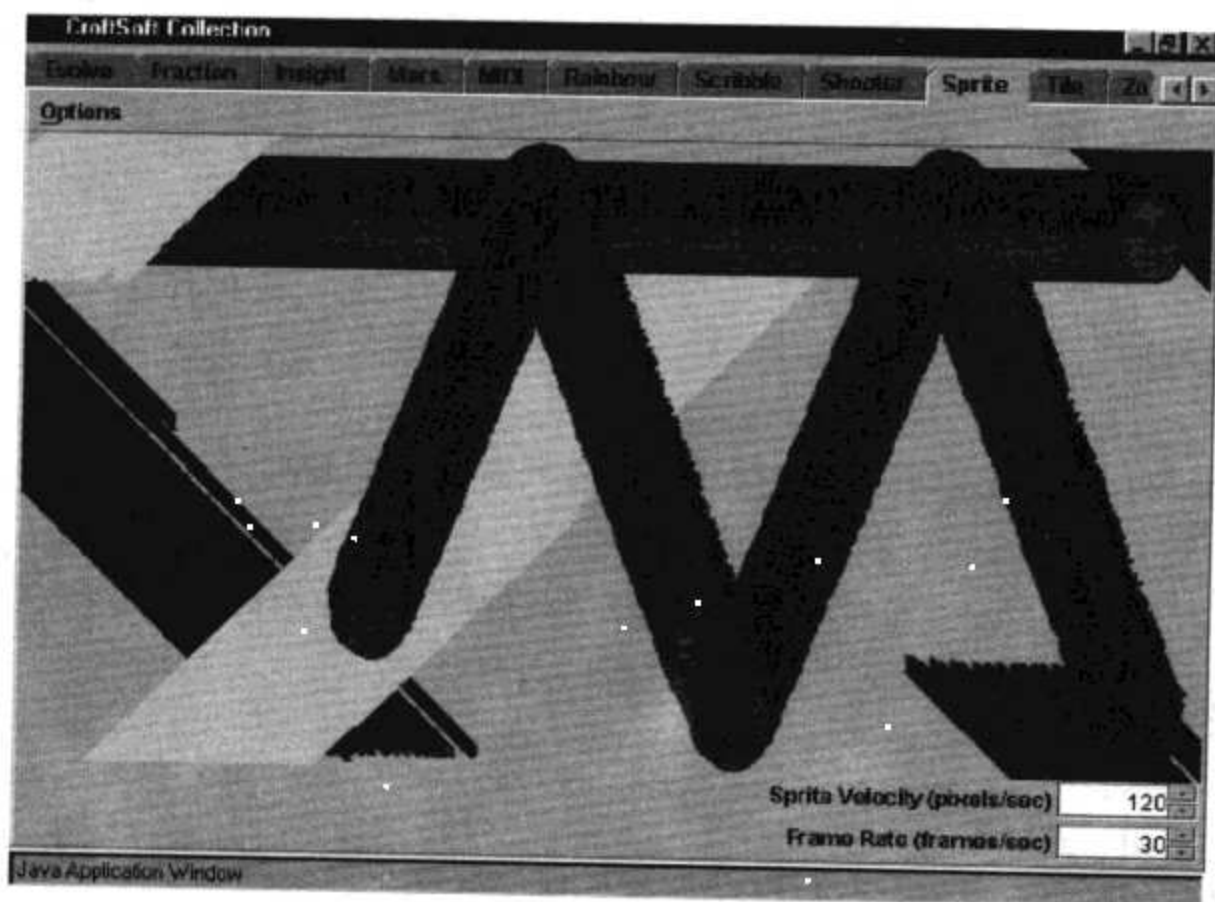


图 3-2 观察动画的平滑性

动画的平滑性主要取决于 `com.croftsoft.core.util.loop` 包中的 `LoopGovernor`。接口 `LoopGovernor` 的具体实现通过让 `animationThread` 休眠一个有限的时间，达到控制帧速率的目的。术语“管理者(Governor)”在动画循环这种背景下，指的是控制机器速度的一个反馈装置。

```
public void govern ( )
    throws InterruptedException;
```

`LoopGovernor` 接口只是定义了一个方法 `Govern()`，这个方法由动画循环进行调用。真正的“魔法”是在其具体实现中发生的。

3.3.1 固定的延迟

`com.croftsoft.core.util.loop` 包的第一个 `LoopGovernor` 实现(即 `FixedDelayLoopGovernor`)，维护着一个周期性的速率，维护的方法是通过一个固定的延迟(以毫秒以及纳秒为单位)，让循环周期性地暂停。

```
package com.croftsoft.core.util.loop;

import com.croftsoft.core.math.MathConstants;

public final class FixedDelayLoopGovernor
```



```

    implements LoopGovernor
    //////////////////////////////////////
    //////////////////////////////////////
    {

    private final long delayMillis;

    private final int delayNanos;

    //////////////////////////////////////
    //////////////////////////////////////

    public FixedDelayLoopGovernor (
        long delayMillis,
        int delayNanos )
    //////////////////////////////////////
    {
        this.delayMillis = delayMillis;

        this.delayNanos = delayNanos;
    }

    public FixedDelayLoopGovernor ( double frequency )
    //////////////////////////////////////
    {
        if ( frequency <= 0.0 )
        {
            throw new IllegalArgumentException ( "frequency <= 0.0" );
        }

        long periodNanos
            = ( long ) ( MathConstants.NANOSECONDS_PER_SECOND / frequency );
        delayMillis
            = periodNanos / MathConstants.NANOSECONDS_PER_MILLISECOND;
        delayNanos = ( int )
            ( periodNanos % MathConstants.NANOSECONDS_PER_MILLISECOND );
    }

```

用一个构造函数接受以 fps 为单位的帧速率，将其作为构造函数参数。再通过求取其帧速率倒数的方法，将其转换为延迟周期。

```

public void govern ( )
    throws InterruptedException
    //////////////////////////////////////
    {
        Thread.sleep ( delayMillis, delayNanos );
    }

```

当固定延迟在更新精灵位置和重绘组件所花费的时间较短的时候，使用一个固定延迟可以起到很好地效果。在这种情况下，由于更新和重绘时间可以忽略，所以循环的总时间将与固定延迟周期大体相等。这种情况通常发生在有一些小而简单的动画时。

如果您不关心实际达到的帧速率，而只关心在给定的机器上动画的连续性的话，使用一个固定的延迟也能够达到很好的效果。在这种情况下，可以将目标帧速率设置到某些大致的范围内。大多数情况下，帧速率为 24~30 就足够维护动画的连续性，但是这可能要依赖帧与帧之间精灵位移距离的大小。您也许希望达到能够达到的最大帧速率，这样无论精灵速度是多少，它的位移始终会是最小。同时，超过显示器的刷新率也是毫无意义的，因为这将会浪费一些处理时间来绘制比显示器能够显示的速率还快的帧。很多人(包括我自己)在显示器的刷新率为 60 赫兹，尤其是超过眼角的时候，可能看到显示器不断地闪烁。因此，显示器刷新率推荐的标准是 85 赫兹。我推荐最大的目标帧速率为 85 赫兹。

3.3.2 帧速率同步

假设将精灵的速度设置到每秒 30 像素，平均帧速率为 24fps。平均帧到帧之间精灵的位移将是每帧 1.25 像素。换句话说，精灵每帧只移动一个像素，但是每四帧移动两个像素。这种周期性位移的矛盾可能值得我们注意。

可以通过将精灵的速度和帧速率进行同步来避免这种矛盾。精灵的速度就会被压缩成每帧多少个整数像素，而不是每帧多少个浮点像素。由于精灵的速度现在与帧速率结合在一起，因此清楚能够在快慢不同的机器上达到的相同帧速率是非常重要的。在这种情况下不能使用 FixedDelayLoopGovernor，因为它只会达到目标帧速率。在较慢的机器上，花费在绘制和更新阶段的时间可能是不能忽略的，所以帧速率可能会剧烈下降。

那么，您所需要的就是 LoopGovernor 的一个实现，这个实现使用一个可变的循环延迟，该延迟的长度正好使您能够不管主机运行的是快还是慢，都正好能够达到理想的帧速率。但是不幸的是，这种实现其实是很难实现的。如果有访问高精度时钟的权限，那么可以从理想的总循环时间中，剔除更新和重绘时间，并为所有其他时间添加延迟。

```
while ( animationIsRunning )
{
    long startTime = System.currentTimeMillis ( );

    update ( );

    repaint ( );

    long finishTime = System.currentTimeMillis ( );

    long elapsedTime = finishTime - startTime;

    long variableDelay = desiredLoopTime - elapsedTime;

    if ( variableDelay < 0 )
    {
        variableDelay = 0;
    }

    Thread.sleep ( variableDelay );
}
```


上面的代码将不会起到什么作用，因为在很多机器上，时钟的精度都是很粗糙的。例如，在 Windows/Intel(Wintel)机器上，时钟精度就总在 50~60 毫秒之间不断跳变。记住，在大约 24fps 左右，动画开始让人眼感觉到很平滑，这要求每帧的显示时间小于 42 毫秒。为了能够分取到 42 毫秒的时间间隔，就需要一个比 50~60 毫秒的精度更高的时钟。

这也可能是不能使用 Swing 的 Timer 类来使动画的帧速率超过 20fps 的原因。Timer 的实现似乎独立于 System 时钟。这是很不幸的，因为它正好比人们可以接受的电影画面质量的 24fps 帧速率小一点。在循环内部使用 Thread.sleep() 似乎能得到更精确的控制。

前面代码不起作用的第二个原因是，它正测量的时间是它请求重绘的时间而不是实际重绘的时间。重新调用 repaint() 方法也只不过是重绘请求进行排队，等待事件分派线程在将来某一个时候进行串行地执行；调用的时候，它实际上并不直接重新绘制组件。

3.3.3 SamplerLoopGovernor

通过调用本地代码而不是使用 System.currentTimeMillis()，可以在 Windows 操作系统上获得高精度的时钟，这样做也意味着出于安全限制的原因，游戏将不能以未签名的 applet 或 Java Web Start 应用程序的形式运行。我曾试过用多种方法以在单纯可移植 Java 中实现帧速率的同步。我曾试用过的一个技术就是设置一个基于帧速率(在相当长的采样时间上计算出的帧速率)的估计延迟间隔。为了能够实现这种技术，可以计算被绘制的帧数，再将这个帧数除以绘制这些帧所花费的时间。使用这种技术，就不用管时钟精度，只要采样周期足够大就可以了。

我在 com.croftsoft.core.util.loop 包的 SamplerLoopGovernor 类中实现了这个技术，使用默认的采样周期 3 秒。为了防止动画帧速率从一个采样时间段到另一个采样时间段时突然增加或突然降低，在下一个采样周期中使用的延迟是新计算的估计目标延迟和在前一个采样时间段中延迟的平均值。

但是，即使采用了这种平滑操作，在采样间隔的长度方面还是有些问题。过长的采样周期可能意味着系统不能快速适应动画复杂程度或可用处理时间上的突然改变。另一方面，如果采样周期过短，在这个周期内计算的帧数就接近 1，估计的精度就会下降。

3.3.4 WindowedLoopGovernor

com.croftsoft.core.util.loop 包的类 WindowedLoopGovernor 通过使用一个称为 windowed averaging 的技术避免了这些问题。与其基于一个固定采样周期估计目标循环的延迟，还不如通过求最近一些帧的测量值的平均数，连续地为每一帧重新估计目标延迟。计算平均值所基于的帧数是窗口(window)。这个窗口也被称为滑窗(slide)，因为时间是流动的，这样在计算中才会只包含最近的测量值；其他的测量值将会被丢弃。

如果平均绘制时间和更新时间发生了突然的变化，延迟将会开始调整到与最近的下一帧相同。这是在 SamplerLoopGovernor 基础上的又一个改进，在 SamplerLoopGovernor 中，直到当前采样周期结束之前，不会进行任何调整。

像 SamplerLoopGovernor 一样，WindowedLoopGovernor 也通过使用平均值来估计延迟。在 SamplerLoopGovernor 中，平均帧速率是通过将帧数除以采样时间得到的。在 WindowedLoopGovernor 中，每一帧的平均更新时间和绘制时间是通过将更新和绘制时间的总和除以采样窗口中的帧数得来的。

```

package com.croftsoft.core.util.loop;

[...]

public final class WindowedLoopGovernor
    implements LoopGovernor
    //////////////////////////////////////
    //////////////////////////////////////
    {

    private static final int DEFAULT_MAX_WINDOW_SIZE = 100;

    private static final long DEFAULT_RESET_TIME_NANOS
        = MathConstants.NANOSECONDS_PER_SECOND;

```

WindowedLoopGovernor 使用一个特殊的技巧：一个可变大小的滑窗。刚开始的时候，滑窗的大小是一帧，然后慢慢地达到一些最大值，默认的最大值是 100 帧。这允许可以立即进行调整而不需要非要等到达到 100 个测量值以后才能进行调整。

如果更新和绘制时间的测量值中有一些值超出了某个范围而变得比较古怪(比默认的一秒的时间还长)，就通过将窗口的大小设为 0 的方法而丢弃这些测量值。这防止平均值受到动画循环中暂停的影响。

```

private final long    periodNanos;

private final int     maxWindowSize;

private final long    resetTimeNanos;

private final long [ ] nonDelayTimes;

```

这个 **final** 变量 **periodNanos** 是理想循环频率的倒数，单位是纳秒。**maxWindowSize** 是用于计算平均测量值的最大帧数。**resetTimeNanos** 是丢弃测量值和重置窗口大小的阈值，单位是纳秒。**nonDelayTimes** 数组保存每一帧更新和绘制时间最新的测量值，直到 **maxWindowSize** 测量值。

```

private int    index;

private int    windowSize;

private long   delayMillis;

private int    delayNanos;

private long   previousTimeNanos;

private long   totalDelayNanos;

private long   sumNonDelayTimes;

```


这个 non-final 实例变量 index 指向 NonDelayTimes 数组中存储测量值的下一个位置。当前 windowSize 的范围是从 0 到 maxWindowSize。totalDelayNanos 数组是用于当前循环中的延迟，也是 delayMillis 和 delayNanos 数组的总和。previousTimeNanos 存储做前面测量的时间，单位是纳秒。sumNonDelayTimes 数组是 NonDelay 数组中所有测量值(这些测量值又是当前窗口的一部分)的总和。

[...]

```
public WindowedLoopGovernor (
    long periodNanos,
    int  maxWindowSize,
    long resetTimeNanos )
    //////////////////////////////////////
{
    if ( periodNanos < 1 )
    {
        throw new IllegalArgumentException ( "periodNanos < 1" );
    }

    this.periodNanos = periodNanos;

    if ( maxWindowSize < 1 )
    {
        throw new IllegalArgumentException ( "maxWindowSize < 1" );
    }

    this.maxWindowSize = maxWindowSize;

    if ( resetTimeNanos < 1 )
    {
        throw new IllegalArgumentException ( "resetTimeNanos < 1" );
    }

    this.resetTimeNanos = resetTimeNanos;

    nonDelayTimes = new long [ maxWindowSize ];

    delayMillis
        = periodNanos / MathConstants.NANOSECONDS_PER_MILLISECOND;

    delayNanos = ( int )
        ( periodNanos % MathConstants.NANOSECONDS_PER_MILLISECOND );

    totalDelayNanos = periodNanos;
}
```

主构造函数将延迟变量初始化为期望的循环周期 periodNanos，这个延迟变量的单位是纳秒。

```
public WindowedLoopGovernor ( double frequency )
    //////////////////////////////////////
{
```

```

    this (
        ( long ) ( MathConstants.NANOSECONDS_PER_SECOND / frequency ),
        DEFAULT_MAX_WINDOW_SIZE,
        DEFAULT_RESET_TIME_NANOS );
}

```

这个便利构造函数使用在循环中给出的期望的循环频率 `frequency`(每秒的频率数)的倒数计算 `periodNanos`。比较合理的 `frequency` 值的范围是 24.0~85.0。该便利构造函数还为 `maxWindowSize` 和 `resetTimeNanos` 参数提供默认值。

```

public void govern ()
    throws InterruptedException
    //////////////////////////////////////
{
    long currentTimeNanos = System.currentTimeMillis ()
        * MathConstants.NANOSECONDS_PER_MILLISECOND;

    long nonDelayTime
        = currentTimeNanos - previousTimeNanos - totalDelayNanos;

    previousTimeNanos = currentTimeNanos;
}

```

对前面一帧进行测量以后所消耗的时间和以前延迟的时间相减,就得到 `nonDelayTime`。由于动画循环被分成 3 个阶段:更新、绘制和延迟,所以假定 `nonDelayTime` 是测量它在最近的帧中,用于完成更新和绘制阶段所花费的时间。

注意,当系统时钟的精度较低时, `currentTimeNanos` 和 `previousTimeNanos` 可能会相同——即使可能有一些时间已经流逝了。在这种情况下,计算得到的 `nonDelayTime` 可能会成为负数。通过将长时间测试的负值和正值进行平均,就会得到一个基本合理的值。

```

long oldNonDelayTime = nonDelayTimes [ index ];

nonDelayTimes [ index ] = nonDelayTime;

sumNonDelayTimes += nonDelayTime;

```

这个新的 `nonDelayTime` 测量值存储在 `nonDelayTimes` 数组中,再将 `sumNonDelayTimes` 加上这个 `nonDelayTime` 值。原来存储在 `index` 位置的 `nonDelayTimes` 就临时保存在 `oldNonDelayTime` 中,当需要从 `sumNonDelayTimes` 中减掉原 `nonDelayTimes` 值的时候,就使用这个 `oldNonDelayTime`。

```

index = ( index + 1 ) % maxWindowSize;

```

该 `index` 是递增的,如果递增到 `maxWindowSize` 就重置为零。

```

if ( nonDelayTime > resetTimeNanos )
{
    windowSize = 0;

    sumNonDelayTimes = 0;
}

```




```

    Thread.sleep ( delayMillis, delayNanos );

    return;
}

```

如果测量的 `nonDelayTime` 比 `resetTimeNanos` 大(默认是 1 秒),就假定游戏循环肯定是被暂停或非正常地停止了。在这种情况下, `nonDelayTime` 中存储的测量值就会被丢弃,丢弃的方法就是将 `windowSize` 和 `sumNonDelayTimes` 重新设置为 0。只能使用以前计算的延迟值,而且该方法会立即返回。

注意,该段代码在首次调用 `govern()` 时总会被调用执行。在这种情况下, `previousTimeNanos` 的初始值就会是 0,因为以前还从来没有调用过这个方法。从 `currentTimeNanos` 中减去 `previousTimeNanos` 这个 0 值就得到自前一帧后所流逝的时间,产生一个非法值。这就会导致产生一个超出范围的 `nonDelayTime`,这个值被上面这个 `if` 语句截取并处理。

```

if ( windowSize == maxWindowSize )
{
    sumNonDelayTimes -= oldNonDelayTime;
}
else
{
    windowSize++;
}

```

如果 `windowSize` 已经增加到等于 `nonDelayTimes` 数组的长度 `maxWindowSize`,新测量的值就会覆盖数组中以前保存的值。但是,在将新测量的值丢弃以前,必须将它们从 `sumNonDelayTimes` 中减去。如果 `windowSize` 比 `maxWindowSize` 小,就一直递增 `windowSize`,而不使用 `oldNonDelayTime`,因为它并没有事先包含在总和中。

```

long averageNonDelayTime = sumNonDelayTimes / windowSize;

totalDelayNanos = periodNanos - averageNonDelayTime;

if ( totalDelayNanos < 0 )
{
    totalDelayNanos = 0;
}

delayMillis
    = totalDelayNanos / MathConstants.NANOSECONDS_PER_MILLISECOND;

delayNanos = ( int )
    ( totalDelayNanos % MathConstants.NANOSECONDS_PER_MILLISECOND );

Thread.sleep ( delayMillis, delayNanos );
}

```

将测量的 `nonDelayTimes` 的总和除以当前的 `windowSize`,就得到 `averageNonDelayTime`。目标延迟时间 `totalDelayNanos` 是从总循环时间 `periodNanos` 中减去 `averageNonDelayTime` 进行估计的。这个线程接着就会延迟 `totalDelayNanos` 的时间,因为 `sleep()` 方法使用了分解开的 `delayMillis` 和 `delayNanos` 作为其参数。

在我看来,在使用低精度系统时钟获取高精度的帧速率同步方面,WindowedLoopGovernor 确实做得非常出色。但是,如果帧到帧之间,更新和绘制的时间变化太大的话,它就可能被欺骗。在这种不正常的情况下,可能必须借助于在游戏中包含本地代码,以在某一种操作系统上,得到高精度的时钟定时。但是,一般情况下,更新和绘制时间的变化会比较合理,可以避免越过默认的安全沙箱限制。

3.4 AnimatedComponent

包 com.croftsoft.core.animation 中的类 AnimatedComponent 是 Swing 组件,该组件通过提供绘制动画的表面、提供调用其他类的动画循环和提供允许将 Swing 组件集成到框架中的生命周期方法,将 Swing 的功能集成到一起。

```
package com.croftsoft.core.animation;

import java.awt.EventQueue;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.lang.reflect.InvocationTargetException;
import javax.swing.JComponent;

import com.croftsoft.core.animation.factory.DefaultAnimationFactory;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.util.loop.LoopGovernor;

public class AnimatedComponent
    extends JComponent
    implements Lifecycle
{
    //////////////////////////////////////
    //////////////////////////////////////
}
```

AnimatedComponent 是一个 Swing 的 JComponent 子类的实现。它实现了 Lifecycle 接口,因此它的动画可以按照需要暂停和重启。

```
public static final String ANIMATION_THREAD_NAME = "Animation";

//

protected final Runnable animationRunner;

//

protected ComponentAnimator componentAnimator;

protected RepaintCollector repaintCollector;

protected LoopGovernor loopGovernor;
```



```
protected Thread      animationThread;

protected boolean     stopRequested;
```

Runnable 类型的实例 animationRunner 用来在一个独立的线程内，调用 update()和 paint()方法；ComponentAnimator 用来激活 AnimatedComponent；RepaintCollector 用来收集重绘请求；LoopGovernor 用来调节动画速度。animationThread 变量指向当前 Thread 的实例，该实例驱动动画循环。布尔标志 stopRequested 用于标识动画循环应该暂停。

接口 ComponentAnimator、RepaintCollector 和 LoopGovernor 实例的引用都不是 final 的，因为它们在运行的过程中可以被替换——如果需要的话。这在 Sprite 示例程序中使用过，目的是比较不同实现之间性能的差别。

```
public AnimatedComponent (
    ComponentAnimator componentAnimator,
    RepaintCollector  repaintCollector,
    LoopGovernor      loopGovernor )
    //////////////////////////////////////
{
    setComponentAnimator ( componentAnimator );

    setRepaintCollector ( repaintCollector );

    setLoopGovernor ( loopGovernor );

    setOpaque ( true );

    animationRunner =
        new Runnable ( )
        {
            public void run ( )
            {
                animate ( );
            }
        };
}
```

方法 setComponentAnimator()、setRepaintCollector()、setLoopGovernor()仅仅保存了构造函数参数的引用，下面再对其进行详细的介绍。

JComponent 方法 setOpaque()用来标志子类的实现是否有某些透明的区域。出于效率方面的考虑，AnimatedComponent 子类会将它自己标志为不透明的，也就是标识为没有透明区域的，因此，在这个组件后面的所有组件和被这个对象模糊化的所有组件都不再需要进行绘制。如果准备将 AnimatedComponent 显示为一个怪异的形状，例如圆圈，就需要将不透明性设置为 false，这样透明区域以后的组件将会按照需要进行重绘。但是，就这种实现来说，我们假定有一个简单的矩形需要显示，并且默认的 JComponent 属性值 false 也被重写了。

构造函数将 final 变量 animationRunner 初始化为一个匿名的内部类的 Runnable 实现，该实现只是转向 protected 方法 animate()。在构造函数中，只会创建一个 animationRunner 实例，因为动画循环会不断重复地使用它而不是为每一个循环创建一个新的实例。

```

public AnimatedComponent (
    ComponentAnimator componentAnimator,
    AnimationFactory animationFactory,
    double frequency)
    //////////////////////////////////////
{
    this (
        componentAnimator,
        animationFactory.createRepaintCollector ( ),
        animationFactory.createLoopGovernor ( frequency ) );
}
public AnimatedComponent (
    ComponentAnimator componentAnimator,
    AnimationFactory animationFactory )
    //////////////////////////////////////
{
    this (
        componentAnimator,
        animationFactory.createRepaintCollector ( ),
        animationFactory.createLoopGovernor ( ) );
}

```

这些便利构造函数都使用了包 `com.croftsoft.core.animation` 中接口 `AnimationFactory` 的一个实例，来创建主构造函数所需要的 `RepaintCollector` 和 `LoopGovernor` 对象。`RepaintCollector` 和 `LoopGovernor` 实现的选取都会明显地影响动画的性能。将 `AnimationFactory` 以一个参数的形式给出，可以选择一个更好的策略。

构造函数中的参数 `frequency` 用来决定动画的速度。电影通过将真实世界的采样快照以 24fps 的速率进行显示，产生连续动作的幻影。通过按照每帧 1/24 秒的时间间隔重绘它自己的方法，`AnimatedComponent` 也可以达到相类似的效果。

第二个便利构造函数没有带频率参数。在这种情况下，将会创建一个 `LoopGovernor`，这个 `LoopGovernor` 是用默认的由 `AnimationFactory` 确定的帧速率运行动画的。

```

public AnimatedComponent (
    ComponentAnimator componentAnimator,
    double frequency )
    //////////////////////////////////////
{
    this (
        componentAnimator,
        DefaultAnimationFactory.INSTANCE,
        frequency );
}

public AnimatedComponent ( ComponentAnimator componentAnimator )
    //////////////////////////////////////
{
    this (
        componentAnimator,
        DefaultAnimationFactory.INSTANCE );
}

```


包 `com.croftsoft.core.animation.factory` 中的 `DefaultAnimationFactory` 是 `AnimationFactory` 的一个推荐实现。由这个工厂产生的具体实现随着新研究的一些更好算法的推出，会不断进行改进。

```
public synchronized ComponentAnimator setComponentAnimator (
    ComponentAnimator componentAnimator )
    //////////////////////////////////////
{
    NullPointerException.check ( componentAnimator );

    ComponentAnimator oldComponentAnimator = this.componentAnimator;

    this.componentAnimator = componentAnimator;

    return oldComponentAnimator;
}

public synchronized RepaintCollector setRepaintCollector (
    RepaintCollector repaintCollector )
    //////////////////////////////////////
{
    NullPointerException.check ( repaintCollector );

    RepaintCollector oldRepaintCollector = this.repaintCollector;

    this.repaintCollector = repaintCollector;

    return oldRepaintCollector;
}

public synchronized LoopGovernor setLoopGovernor (
    LoopGovernor loopGovernor )
    //////////////////////////////////////
{
    NullPointerException.check ( loopGovernor );

    LoopGovernor oldLoopGovernor = this.loopGovernor;

    this.loopGovernor = loopGovernor;

    return oldLoopGovernor;
}
```

这些 `mutator` 方法在动画过程中，允许对 `ComponentAnimator`、`RepaintCollector` 和 `LoopGovernor` 的实例按照需要进行替换。这些方法返回替换后的实例以备后面使用。这些方法都被同步，以防止它们被多个线程同步进行调用。如果假定很少调用这些方法，平均每个动画循环调用还不到一次，那同步这些方法的性能开销就可以忽略。

多个 `ComponentAnimator`、`RepaintCollector` 和 `LoopGovernor` 的实例可以在内存中进行预构造，再立即按照需要交换 `AnimatorComponent`。例如，交替 `ComponentAnimator` 可能会导致

场景突然切换到一个不同的视图中，例如突然切换到动画的暂停屏幕，然后又突然返回到正常的屏幕。

如果参数为空，静态方法 `check()` 就抛出 `NullPointerException` 异常。由于这个原因，`ComponentAnimator`、`RepaintCollector` 和 `LoopGovernor` 实例变量永远不可能被赋上空值。明白这一点以后，在频繁调用的动画操作中，就没有必要去检查空引用了。当没有其他实例可用或适用的时候，像 `NullRepaintCollector` 这样的类作为临时空操作的占位符(或空对象)都是很有用的。

```
public void init ( )
///////////////////////////////////////////////////
{
}
```

这个 `init()` 方法是个空方法。子类实现可能重写这个方法而不用去调用其父类的实现，但是万一将来的版本是一个非空的方法，子类的实现也许还必须去调用其父类的实现。

通用的生命周期方法和下面使用的动画线程管理的具体技术在前一章中都进行过介绍。在继续阅读本书之前，建议读者复习一下前面的内容。

```
public synchronized void start ( )
///////////////////////////////////////////////////
{
    stopRequested = false;

    if ( animationThread == null )
    {
        animationThread = new Thread (
            new Runnable ( )
            {
                public void run ( )
                {
                    loop ( );
                }
            },
            ANIMATION_THREAD_NAME );

        animationThread.setPriority ( Thread.MIN_PRIORITY );

        animationThread.setDaemon ( true );

        animationThread.start ( );
    }
    else
    {
        notify ( );
    }
}
```

在首次调用 `start()` 方法的时候，它会创建并启动一个 `animationThread` 方法。在暂停以后，再次调用 `start()` 方法，重新启动动画的时候，它就重新将 `stopRequested` 的标志设为 `false`，并通知动画循环线程应该重新恢复。

线程的优先级被设置为最低，目的是防止动画线程阻塞其他 Swing 事件。假定动画是在一个快速循环中连续运行的，那么将它设为一个较低的权限对于防止用户体验由于鼠标和键盘输入的延迟处理而带来的响应迟缓而言也是非常必要的。

`setDaemon()`方法的调用表示 `animationThread` 将用作 daemon 线程。daemon 线程与正常线程的不同在于，它在全部正常线程终止以后自动终止。这使它们对进行 fire-and-forget 背景处理来说非常理想。通过指定 daemon 线程的使用，`AnimatorComponent` 保证了容器程序可以完成，而不需要对 `animationThread` 进行显式的控制。

```
public synchronized void stop ( )
///////////////////////////////////////////////////
{
    stopRequested = true;

    animationThread.interrupt ( );
}
```

`stop()`方法设置标志 `stopRequested`。像下面描述的一样，这将会导致动画循环暂停。它也中断 `animationThread`，让 `animationThread` 暂停它正在处理的事情，并检查 `stopRequested` 标志的状态。

```
public synchronized void destroy ( )
///////////////////////////////////////////////////
{
    animationThread = null;

    stopRequested = false;

    notify ( );
}
```

`destroy()`方法解除对 `animationThread` 的引用，这将导致动画循环在下一次循环迭代开始的地方终止。为了让循环在被挂起的时候，能够到达下一个循环迭代的开始处，它必须通过重新将 `stopRequested` 标志设为 `false` 的方法进行重启，并产生一个通告。

```
public void paintComponent ( Graphics graphics )
///////////////////////////////////////////////////
{
    componentAnimator.paint ( this, ( Graphics2D ) graphics );
}
```

这个方法只是将绘制操作委托给 `ComponentAnimator`。`paintComponent()`方法重写了其超类 `JComponent` 的方法。在基于 AWT(Abstract Window Toolkit)的动画中，`Component` 的子类会重写 `paint()`方法，以提供定制的代码来绘制组件的表面。但是，在 Swing `JComponent` 子类中，方法 `paint()`还负责绘制组件的边框和子组件(如果有子组件的话)。它是通过依次调用 `paintComponent()`、`paintBorder()`和 `paintChildren()`方法来实现的。正是由于这个原因，`JComponent` 的子类 `AnimatedComponent` 才重写 `paintComponent()`方法而不是 `paint()`方法。

由于通常都不是必要的，所以 `AnimatedComponent` 的 `paintComponent()`方法就不在委托给 `ComponentAnimated` 的 `paint()`方法之前，使用背景颜色填充组件矩形。例如绘制横穿整个组件区域的不透明背景图像的 `ComponentAnimated` 就只是简单地覆盖以前绘制的所有帧，这使得清

除表面的所有工作都变得毫无意义。JComponent 超类方法 paintComponent()实现, 采用一种子类最方便实现的方式, 清除组件的表面。然而在假定这个第一步通常都不必要的时候, 在 AnimatedComponent 子类的实现中就不会调用 super.paintComponent()方法。AnimatedComponent 转而依赖 ComponentAnimator 的实例, 完全重绘组件区域以覆盖所有原来的像素——如果需要覆盖的话。

注意, 在动画循环暂停的时候, 也可能调用 paintComponent()方法。在这种情况下, 只需要刷新屏幕而不会更新精灵的位置。例如, 游戏暂停的时候, 如果窗口改变大小或被其他的窗口覆盖以后又移开, 就会发生这种情况。

```
public void repaint ( )
///////////////////////////////////////////////////
{
    repaintCollector.repaint ( );
}

public void repaint ( long tm )
///////////////////////////////////////////////////
{
    repaintCollector.repaint ( );
}
```

这些重写的 JComponent 重绘方法将重绘整个组件的请求委托给 RepaintCollector。

```
public void repaint (
    int x,
    int y,
    int width,
    int height )
///////////////////////////////////////////////////
{
    repaintCollector.repaint ( x, y, width, height );
}

public void repaint (
    long tm,
    int x,
    int y,
    int width,
    int height )
///////////////////////////////////////////////////
{
    repaintCollector.repaint ( x, y, width, height );
}

public void repaint ( Rectangle r )
///////////////////////////////////////////////////
{
    repaintCollector.repaint ( r.x, r.y, r.width, r.height );
}
```

这 3 个重绘方法与前面两个重绘方法相似, 只是请求重绘的是一个较小的区域而不是整个组件的表面。这可以使性能得到显著地提高。


```

protected void loop ( )
////////////////////////////////////
{
    while ( animationThread != null )
    {
        try
        {
            EventQueue.invokeAndWait ( animationRunner );
            loopGovernor.govern ( );
            if ( stopRequested )
            {
                synchronized ( this )
                {
                    while ( stopRequested )
                    {
                        wait ( );
                    }
                }
            }
        }
        catch ( InterruptedException ex )
        {
        }
        catch ( InvocationTargetException ex )
        {
            ex.getCause ( ).printStackTrace ( );
        }
    }
}

```

loop()方法是 AnimatedComponent 的“心脏”——它就是保持动画心跳的方法。它周期性地 将 AnimatedRunner 在事件分派线程内部排队执行，再将动画循环延迟足够长的时间，以达到理想的帧速率。

由于 stop()方法也在 animationThread 上调用 Thread.interrupt()方法，所以 govern()方法检查 Thread.interrupt()状态的时候，就可能会引发 InterruptedException 异常。当在 govern()方法实现的内部调用了 Thread.sleep()方法，这种检查经常会自动发生。

通过 destroy()方法解除 animationThread 引用的时候，循环就会终止。注意，当调用 destroy()方法的时候，它不会立即终止，而是在下一个循环迭代开始处进行检查时终止。如果子类的实现不需要等到循环完成以后再延迟释放在游戏中用到的资源，这可能会导致问题的复杂化。既然 destroy()方法很可能在 loop()方法之前完成，那么这些资源在 loop()方法的结束处也应该被释放。这可以通过在子类的实现中重写 loop()方法来实现。

```

protected void animate ( )
////////////////////////////////////
{
    componentAnimator.update ( this );

    int count = repaintCollector.getCount ( );

```

```

Rectangle [ ] repaintRegions
    = repaintCollector.getRepaintRegions ( );

for ( int i = 0; i < count; i++ )
{
    paintImmediately ( repaintRegions [ i ] );
}

repaintCollector.reset ( );
}

```

这个私有的 `animate()` 方法首先调用 `ComponentAnimator` 的 `update()` 方法，这个方法更新屏幕上精灵的位置，只要需要，它还请求 `AnimatedComponent` 重绘它自身。当所有精灵的位置都已经被更新，所有的重绘请求也已经被收集起来以后，就为每一个重绘区域调用 `paintImmediately()` 方法。该 `paintImmediately()` 方法再调用 `paintComponent()` 方法，后者被重写以调用 `ComponentAnimator` 的 `paint()` 方法。无论直接或者是间接，`animate()` 的任务就是按照那个顺序调用 `ComponentAnimator` 的 `update()` 方法和 `paint()` 方法。完全以后，它就为下一个动画循环重置 `repaintCollector`。

动画循环暂停的时候，就不再调用 `animate()` 方法，由重写的 `repaint()` 方法发起的所有发给 `repaintCollector` 的请求还会继续聚集，只是不再进行处理。幸运地是，需要重绘组件的系统窗口事件会绕过 `repaint()` 方法，并直接调用 `paintComponent()`。这允许屏幕在游戏被暂停的同时进行重绘，而不需要向 `repaintCollector` 发送重绘请求。

既然 `animate()` 方法只在事件分派线程中进行调用，那么更新、重绘和重置操作都被适当地串行化了。有人可能会担心由应用程序从动画循环以外的其他地方产生的对 `repaint()` 方法的调用，可能会在两个方面产生问题。首先，在精灵的位置正在被更新的时候，可能会试图重绘屏幕。第二个方面是，如果正好在调用 `repaintCollector` 的 `reset()` 方法以前收集到重绘请求，那么该重绘请求就可能会被丢失。然而，这两个问题实际上都不可能成为问题，因为在事件分派线程中，重绘请求都是串行处理的。

3.5 小结

本章主要介绍了 4 个基本的动画类：`ComponentAnimator`、`RepaintCollector`、`LoopGovernor` 和 `AnimatedComponent`。接口 `ComponentAnimator` 的实现提供了应用程序特定的大部分代码，因为它给出了更新精灵位置的游戏逻辑。作为一般用途的 `RepaintCollector` 和 `LoopGovernor` 实现以及 `AnimatedComponent` 组成了基于 Swing 动画引擎可重用的核心。掌握这些类的操作和互操作对于调整动画的性能而言是至关重要的。

3.6 参考文献

Robinson, Matthew and Pavel Vorobiev, "Swing Mechanics." Chapter 2 in *Swing*. Greenwich, CT: Manning Publications Company, 2000.

第 4 章 动 画 库

流逝的时间是再也找不回来的东西。

——本杰明·富兰克林

前一章将 `AnimatedComponent` 作为游戏引擎的心脏进行介绍，因为这个组件提供了驱动动画的游戏循环的心跳机制。我喜欢将 `AnimatedComponent` 接口的实现作为游戏的大脑，因为它们提供游戏特定的逻辑——决定怎样在屏幕上移动对象和怎样响应用户的输入。

回想一下前面的内容，`ComponentAnimator` 扩展了 `ComponentUpdater` 和 `ComponentPainter`，分别提供了 `update()` 方法和 `paint()` 方法。这也就提出了一种在可重用动画库中将类分组的方式。有些类实现了 `ComponentUpdater`，有些实现了 `ComponentPainter`，还有一些类实现了 `ComponentAnimator`，而后者又扩展了前两个组件。本章的最后一个内容，即接口 `Sprite`，就是一个可重用 `ComponentAnimator` 的扩展，该扩展为在虚拟世界中移动的游戏实体提供了通用方法。

4.1 ComponentPainter 实现

可以在 `com.croftsoft.core.animation.painter` 包中找到许多很有用的 `ComponentPainter` 实现。这里只对它们作一个很简单的说明，本节稍后会对其进行更为详细的介绍。`NullComponentPainter` 什么事也不做，只是纯粹地作为一个占位符。`ArrayComponentPainter` 是一个复合对象，该对象使 `ComponentPainter` 的多个实例看起来好像是一个实例。`ColorPainter` 使用 `Color` 绘制组件。`SpacePainter` 绘制太空的背景。`TitlePainter` 绘制 `Icon` 的网格。

4.1.1 NullComponentPainter

`NullComponentPainter` 是 `ComponentPainter` 的一个空对象单态实现。在调用它的 `paint()` 方法时，它其实不做任何什么事情。它只是作为一个占位符，防止产生 `NullPointerException` 异常。

4.1.2 ArrayComponentPainter

类 `ArrayComponentPainter` 是一个复合对象，它使 `ComponentPainter` 的一组实例看起来就像是一个 `ComponentPainter` 实例。由于在一组 `ComponentPainter` 实例上通过循环迭代的方式逐一绘制每个实例是一种很常用的操作，所以类 `ArrayComponentPainter` 用一个可重用的方式封装了这种逻辑。

```
package com.croftsoft.core.animation.painter;
```

```

import java.awt.Graphics2D;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentPainter;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.util.ArrayLib;

public final class ArrayComponentPainter
    implements ComponentPainter
    //////////////////////////////////////
    //////////////////////////////////////
{

    private ComponentPainter [ ] componentPainters;

    //////////////////////////////////////
    // constructor methods
    //////////////////////////////////////

    public ArrayComponentPainter (
        ComponentPainter [ ] componentPainters )
    //////////////////////////////////////
    {
        setComponentPainters ( componentPainters );
    }

    public ArrayComponentPainter ( )
    //////////////////////////////////////
    {
        this ( new ComponentPainter [ 0 ] );
    }

    //////////////////////////////////////
    // accessor/mutator methods
    //////////////////////////////////////
    public ComponentPainter [ ] getComponentPainters ( )
    //////////////////////////////////////
    {
        return componentPainters;
    }

    public void add ( ComponentPainter componentPainter )
    //////////////////////////////////////
    {
        componentPainters = ( ComponentPainter [ ] )
            ArrayLib.append ( componentPainters, componentPainter );
    }

    public void setComponentPainters (
        ComponentPainter [ ] componentPainters )
    //////////////////////////////////////
    {
        NullArgumentException.check (
            this.componentPainters = componentPainters );
    }

```



```

}

////////////////////////////////////
////////////////////////////////////

public void paint (
    JComponent component,
    Graphics2D graphics )
{
    //////////////////////////////////////
    {
        for ( int i = 0; i < componentPainters.length; i++ )
        {
            componentPainters [ i ].paint ( component, graphics );
        }
    }
}

```

通过调用 `ArrayComponentPainter` 类的 `paint()` 方法, 就相当于调用了 `componentPainters` 了数组中每一个 `ComponentPainter` 实例的 `paint()` 方法。有一点需要注意, 先添加到 `ArrayComponentPainter` 中的 `ComponentPainter` 实例会比后添加到 `ArrayComponentPainter` 中的 `ComponentPainter` 实例先绘制(也就是先来先绘制的方式)。正是由于这个原因, 我们才应该先添加背景 `ComponentPainter` 实例, 再添加前景 `ComponentPainter` 实例。如果有些 `ComponentPainter` 实例好像没有绘制, 那么可能是由于它们的像素在绘制以后立即被其他实例覆盖了。

由于 `ComponentAnimator` 扩展了 `ComponentPainter`, 所以 `componentPainters` 数组中有些实例可能实现了 `ComponentAnimator`。另外, 由于 `ArrayComponentPainter` 实现了 `ComponentAnimator`, 所以 `ArrayComponentPainter` 可能包含其他 `ArrayComponentPainter` 的实例。这就允许我们可以用层次结构的方式组织复合 `ComponentPainter`。

4.1.3 ColorPainter

`ColorPainter` 用实心的 `Color` 绘制 `JComponent` 的表面。

```

package com.croftsoft.core.animation.painter;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Shape;
import java.io.Serializable;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentPainter;

public final class ColorPainter
    implements ComponentPainter, Serializable
{
    //////////////////////////////////////
    //////////////////////////////////////

    private static final long serialVersionUID = 0L;

```

```
//

private Color color;

private Shape shape;

////////////////////////////////////
////////////////////////////////////

public ColorPainter (
    Color color,
    Shape shape )
////////////////////////////////////
{
    this.color = color;

    this.shape = shape;
}
```

如果构造函数的 `color` 参数为空，那么就会使用组件的背景颜色。如果 `shape` 为空，就会重新绘制整个组件。如果想使用背景颜色为整个组件着色，将 `shape` 设为空是非常有用的。如果 `color` 是一个半透明的颜色，那么它就可以用来为前面绘制的任意像素着色。例如，半透明的黑色可以将场景变暗，半透明的白色则可以为场景添加雾气。

```
[...]

public void paint (
    JComponent component,
    Graphics2D graphics )
////////////////////////////////////
{
    if ( color == null )
    {
        graphics.setColor ( component.getBackground ( ) );
    }
    else
    {
        graphics.setColor ( color );
    }

    if ( shape == null )
    {
        graphics.fillRect ( 0, 0, Integer.MAX_VALUE, Integer.MAX_VALUE );
    }
    else
    {
        graphics.fill ( shape );
    }
}
```

注意，可以用实心的颜色填充所选择的所有图形。这包括基本的图形(例如矩形和圆形)和复杂的图形(例如自己定义的任意多边形)。

4.1.4 SpacePainter

类 `SpacePainter` 是 `ComponentPainter` 的一个示例实现。它用黑色的背景加上一些星星填充组件的区域。只要想将黑夜或太空作为背景,它就可以是一个很有用的类。图 4-1 是 `SpacePainter` 类在 CroftSoft Collection 中 applet Shooter 中应用的示例。

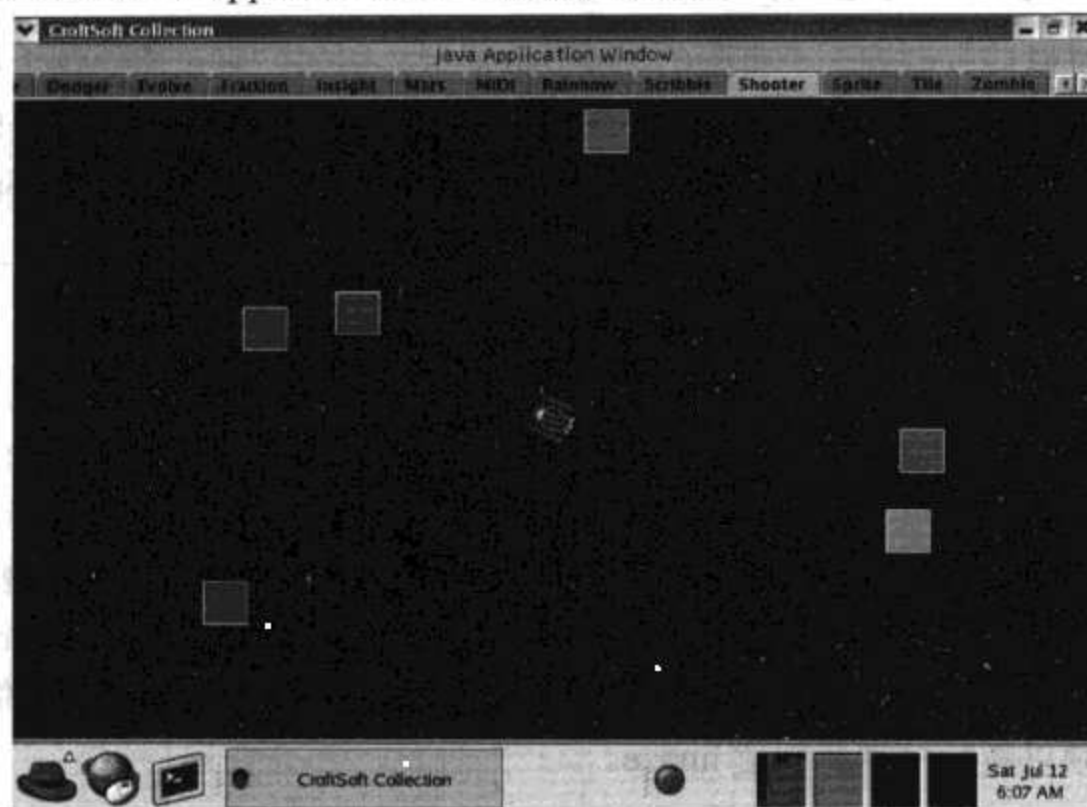


图 4-1 `SpacePainter` 示例

```
package com.croftsoft.core.animation.painter;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.Rectangle;
import java.util.Random;
import javax.swing.JComponent;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.animation.ComponentPainter;
public final class SpacePainter
    implements ComponentPainter
{
    ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    private static final double DEFAULT_STAR_DENSITY = 0.01;

    /**
     * Probability of a star at any given pixel.
     */
    private final double starDensity;

    private final Random random;

    private final long seed;
```

星星在每一个像素上出现概率的默认值为百分之一。将随机数生成器的种子(seed)保存起来,以便当屏幕的大小改变的时候,能够重新产生这些星星。

```
private Rectangle  paintArea;

private boolean    useComponentBounds;
```

空间的绘制可以被限制到一个有限的 `paintArea` 上。例如,假设想绘制一块星星在远处山头上闪烁的背景区域,就可以将 `paintArea` 限制到屏幕的上半部分,因为屏幕的下半部分要用来绘制山峰。如果将 `useComponentBounds` 设为 `true`,那么整个组件都将会由 `SpacePainter` 进行重绘。

```
private Image      image;

private int        width;

private int        height;
```

必须为每一个像素产生一个随机数的时候,就会需要花一段时间产生这些星星。如果像这样绘制每一帧的时候都产生这些星星,还不如只产生一次,并在内存中将背景作为一个 `image` 保存起来。保存后,通过复制 `image` 数据的方法就可以执行绘制了。当组件的 `width` 或 `height` 改变的时候,必须以新尺寸重新创建 `image`。

```
public SpacePainter (
    Rectangle paintArea,
    double starDensity )
    //////////////////////////////////////
{
    setPaintArea ( paintArea );

    this.starDensity = starDensity;

    random = new Random ( );

    seed = random.nextLong ( );
}

public SpacePainter ( )
    //////////////////////////////////////
{
    this ( null, DEFAULT_STAR_DENSITY );
}
```

调用不带参数的便利构造函数会创建一个 `SpacePainter` 实例,该实例使用星星的默认密度来绘制整个组件。

```
public void setPaintArea ( Rectangle paintArea )
    //////////////////////////////////////
{
    useComponentBounds = ( paintArea == null );
}
```



```

    if ( useComponentBounds )
    {
        this.paintArea = new Rectangle ( );
    }
}

```

如果 `paintArea` 为空，就会重新绘制整个组件。在这里，`paintArea` 变量就存储组件的边界。

```

public void paint (
    JComponent component,
    Graphics2D graphics )
////////////////////////////////////////////////////
{
    if ( useComponentBounds )
    {
        component.getBounds ( paintArea );

        if ( ( width != paintArea.width )
            || ( height != paintArea.height ) )
        {
            if ( image != null )
            {
                image.flush ( );

                image = null;
            }
        }
    }

    if ( image == null )
    {
        width = paintArea.width;

        height = paintArea.height;

        createImage ( component );
    }

    graphics.drawImage ( image, paintArea.x, paintArea.y, null );
}

```

如果这是第一次调用 `paint()` 方法或 `component` 的大小已经发生了改变，原来的所有 `image` 中的绝大部分都将会被销毁，在绘制新的 `image` 以前，当然就要重新产生这个新的 `image`。

```

private void createImage ( JComponent component )
////////////////////////////////////////////////////
{
    image = component.createImage ( width, height );

    Graphics graphics = image.getGraphics ( );

    graphics.setColor ( Color.BLACK );
}

```

```
graphics.fillRect ( 0, 0, width, height );

random.setSeed ( seed );

graphics.setColor ( Color.WHITE );

for ( int x = 0; x < width; x++ )
{
    for ( int y = 0; y < height; y++ )
    {
        if ( random.nextDouble ( ) <= starDensity )
        {
            graphics.fillOval ( x, y, 1, 1 );
        }
    }
}
```

私有方法 `createImage()` 创建一个理想宽度和高度的图片。整个图片会被绘制成黑色。然后 `createImage()` 迭代每一个像素，确定是否要将像素变为星星。按照一个像素的大小绘制星星的长和宽。

4.1.5 TilePainter

类 `TilePainter` 用瓦片状或网格状图案在组件上绘制 `Icon`。`TilePainter` 非常灵活，还可以将它用作场景管理器(scene manager)。玩家的图标在游戏视图中移动时，场景管理器就管理背景动画的切换。图 4-2 是 `TilePainter` 类在 `CroftSoft Collection` 的 applet `Title` 中应用的示例。

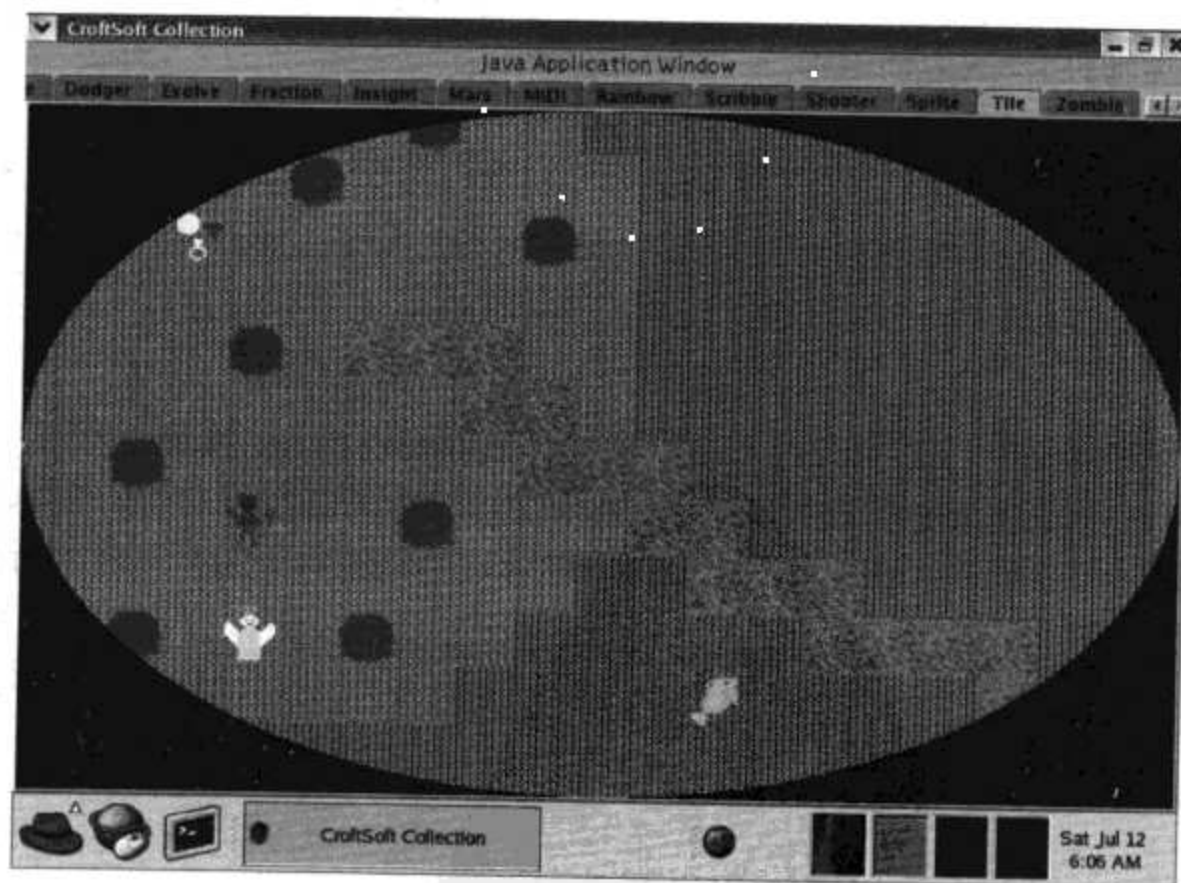


图 4-2 `TilePainter` 示例


```

package com.croftsoft.core.animation.painter;

import java.awt.*;
import java.awt.geom.Rectangle2D;
import javax.swing.*;

import com.croftsoft.core.animation.ComponentPainter;
import com.croftsoft.core.lang.NullArgumentException;

public final class TilePainter
    implements ComponentPainter
    //////////////////////////////////////
    //////////////////////////////////////
{

    private final Shape      tileShape;

    private final int        tileWidth;

    private final int        tileHeight;

    private final Icon [ ]    tileIcons;

    private final byte [ ] [ ] tileMap;

    private final Rectangle   clipBounds;

    private final Rectangle   originalClipBounds;

    tileShape 是要在上面绘制瓦片(tile)的组件表面中的一个区域。tileWidth 是每个瓦片的宽度，
    单位是像素；tileHeight 是每个瓦片的高度，单位也是像素。tileIcons 数组保存 Icon 实例。tileMap
    是一个二维的地图，标明在虚拟的世界中瓦片的移动方向。clipBounds 为图上下文保存调整过的
    剪切区域。originalClipBounds 保存原始的没有经过剪切的区域，以便能够进行恢复。

    private int offsetX;

    private int offsetY;

```

非 final 的实例变量 offsetX 和 offsetY 用于在水平和垂直两个方向上移动瓦片图案。例如，在 CroftSoft Collection 的 applet Sprite 中，就使用了一个描述墙砖图案的瓦片 Icon。offsetX 和 offsetY 是以每一帧中要改变一个像素的方式修改的，这样就产生了背景滑动的效果。

```

public TilePainter (
    int            offsetX,
    int            offsetY,
    Icon [ ]       tileIcons,
    byte [ ] [ ]   tileMap,
    Dimension      tileSize,
    Shape          tileShape )
    //////////////////////////////////////
{

```

```

    this.offsetX = offsetX;

    this.offsetY = offsetY;

    NullPointerException.check ( this.tileIcons = tileIcons );

    NullPointerException.check ( this.tileMap = tileMap );

    if ( tileIcons.length < 1 )
    {
        throw new IllegalArgumentException ( "tileIcons.length < 1" );
    }

    if ( tileIcons.length > 256 )
    {
        throw new IllegalArgumentException ( "tileIcons.length > 256" );
    }

```

由于 `tileMap` 是以一组字节数组的形式存储的，所以 `tileIcons` 数组的长度不能大于 256。`TileMap` 字节值作为 `tileIcons` 数组的索引。例如，如果在 `tileMap` 里给定的两维单元格位置中的字节值为 1，就意味将要使用 `tileIcons` 数组中第 2 个 Icon。

```

for ( int i = 0; i < tileIcons.length; i++ )
{
    if ( tileIcons [ i ] == null )
    {
        throw new IllegalArgumentException (
            "tileIcons[" + i + "] == null" );
    }
}

```

`tileIcons` 数组中不能有空元素。

```

if ( tileMap.length < 1 )
{
    throw new IllegalArgumentException ( "tileMap.length < 1" );
}

```

`tileMap` 必须至少有一行值。

```

int tilesWide = tileMap [ 0 ].length;

if ( tilesWide < 1 )
{
    throw new IllegalArgumentException ( "tileMap[0].length < 1" );
}

```

`tileMap` 必须至少有一列值。

```

for ( int row = 0; row < tileMap.length; row++ )
{
    if ( tileMap [ row ].length != tilesWide )
    {

```



```

        throw new IllegalArgumentException (
            "tileMap[" + row + "].length != tileMap[0].length" );
    }

```

tileMap 的每一行的列数都必须相同。

```

for ( int column = 0; column < tileMap [ row ].length; column++ )
{
    int paletteIndex = 0xFF & tileMap [ row ] [ column ];

    if ( paletteIndex >= tileIcons.length )
    {
        throw new IllegalArgumentException (
            "tileMap[" + row + "][" + column + "] >= tileIcons.length" );
    }
}

```

tileMap 中的每一个 paletteIndex 值都必须指向 tileIcons 数组中一个有效的位置。注意掩码操作将 -128~+127 之间的字节值转换为 0~255 之间的一个整数索引位置。

```

if ( tileSize == null )
{
    tileWidth = tileIcons [ 0 ].getIconWidth ( );

    tileHeight = tileIcons [ 0 ].getIconHeight ( );
}
else
{
    tileWidth = tileSize.width;

    tileHeight = tileSize.height;
}

```

如果 tileSize 为空，tileIcons 数组中第一个 Icon 的宽和高就用来隔开单元格。如果 tileSize 的大小与实际 Icon 的宽和高的值不一样，那么瓦片就可能会被分隔得太远或重叠在一起。如果想创建等轴视图，重叠瓦片是非常有用的。

```

if ( ( tileWidth < 1 )
    || ( tileHeight < 1 ) )
{
    throw new IllegalArgumentException (
        "tileWidth < 1 or tileHeight < 1" );
}

this.tileShape = tileShape;

clipBounds = new Rectangle ( );

originalClipBounds = new Rectangle ( );
}

```

tileWidth 和 tileHeight 的值都必须大于 0。如果 tileShape 为空，整个组件都将使用瓦片重绘。Final 变量 clipBounds 和 originalClipBounds 在构造函数中只初始化一次，然后就重复使用它们来存储更新后的值。

```
public TilePainter (
    int    offsetX,
    int    offsetY,
    Icon    icon,
    Shape tileShape )
    //////////////////////////////////////
{
    this (
        offsetX,
        offsetY,
        new Icon [ ] { icon },
        new byte [ ] [ ] { { 0 } },
        ( Dimension ) null,
        tileShape );
}
```

当只想使用一个 icon 和只有一行一列的 tileMap 时，就需要使用这个便利构造函数。一般地，会将场景分割为多个瓦片以提高效率，但是也可能会遇到要将一大张图片作为背景的情况。图片的大小可能会超出可视区域。offsetX 和 offsetY 的值就用来决定实际可见场景的哪一块区域。tileShape 可以更进一步地限制可见性。

例如，假定有一张高清晰的海底风景图片。无论在什么时候，通过潜艇的舷窗就只能看到这个图片的某一部分。tileShape 将显示的区域限制为圆形潜艇窗口的大小。在任何给定时间，当潜艇沿着海底移动的时候，这张大图片的哪一部分可见是由 offsetX 和 offsetY 的值决定的。

[...]

```
public void setOffsetX ( int offsetX ) { this.offsetX = offsetX; }
```

```
public void setOffsetY ( int offsetY ) { this.offsetY = offsetY; }
```

这些 mutator 方法用来改变偏移的值。在前面列出的代码中，还省略了另外一个便利构造函数和许多 accessor 方法。

```
public int getTileRow ( Point mousePoint )
    //////////////////////////////////////
{
    int row = floorDivision ( mousePoint.y, offsetY, tileHeight );

    row = row % getTileRows ( );

    if ( row < 0 )
    {
        row += getTileRows ( );
    }
    return row;
}

[...]
```


在 CroftSoft Collection 中的 applet Title 中，用户可以通过单击瓦片的方法来更改瓦片。方法 `getTileRow()` 和 `getTileColumn()` 用来将以像素为单位的 `mouse Point` 的 `x` 和 `y` 值，转换为以瓦片单元格为单位的行和列的值。由于瓦片可以重复和回绕整个屏幕，计算的行值可能必须进行标准化处理，也就是进行转换，以使它能在一个正常的值范围以内。`getTileColumn()` 的代码与 `getTileRow()` 的代码很相似。

```
public void paint (
    JComponent component,
    Graphics2D graphics )
    //////////////////////////////////////
{
    graphics.getClipBounds ( clipBounds );

    if ( tileShape != null )
    {
        if ( !tileShape.intersects ( clipBounds ) )
        {
            return;
        }

        graphics.setClip ( tileShape );

        originalClipBounds.setBounds ( clipBounds );

        Rectangle2D.intersect (
            originalClipBounds, tileShape.getBounds2D ( ), clipBounds );
    }
}
```

如果矩形 `clipBounds` 没有和 `tileShape` 重叠，`paint()` 方法就不做任何事情。否则剪裁的区域就被设为 `tileShape`，因为要被铺上瓦片的区域可能会是一个不规则的形状，例如圆形。将 `originalClipBounds` 存储起来，以便当 `paint()` 方法返回的时候，能够将它们恢复。经过调整的 `clipBounds` 减缩到与 `tileShape` 重叠的更小的矩形区域。图 4-3 中用交叉线标出的区域就是实际要在屏幕上绘制的区域。注意，如果 `tileShape` 为空，就使用原来未经过调整的 `clipBounds` 来确定要被铺上瓦片的区域的大小。

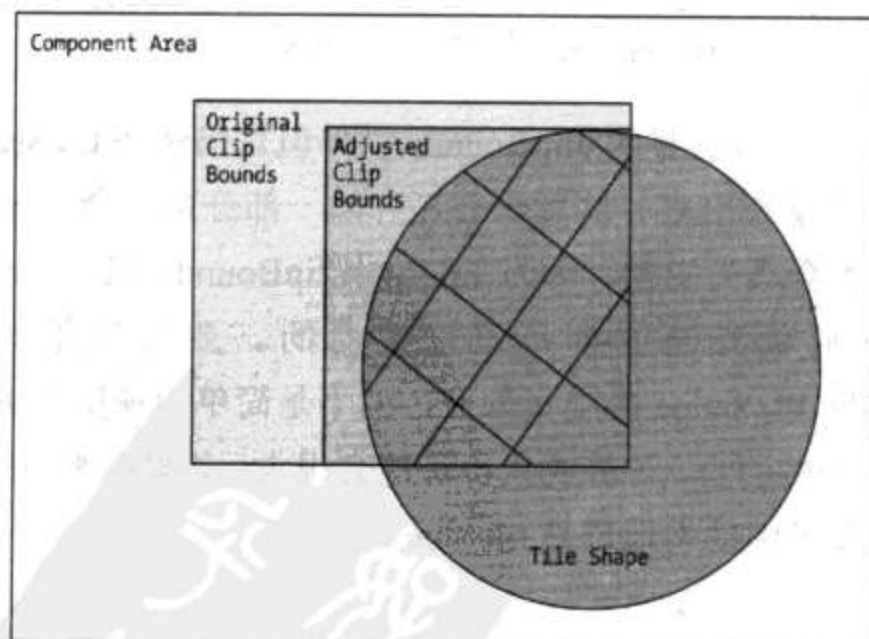


图 4-3 调整以后的剪裁区域

```
int minX = clipBounds.x;  
int maxX = clipBounds.x + clipBounds.width - 1;  
int minY = clipBounds.y;  
int maxY = clipBounds.y + clipBounds.height - 1;  
int minColumn = floorDivision ( minX, offsetX, tileWidth );  
int maxColumn = floorDivision ( maxX, offsetX, tileWidth );  
int minRow = floorDivision ( minY, offsetY, tileHeight );  
int maxRow = floorDivision ( maxY, offsetY, tileHeight );
```

只要是与 clipBounds 重叠的瓦片，不管是调整过的还是没有调整过的，都会被绘制出来。clipBounds 的最大和最小值(单位是像素)，都被转换为瓦片的最大和最小的行和列的索引值。图 4-4 显示了瓦片与调整的 clipBounds 重叠，以及表示 tileShape 区域的圆环。

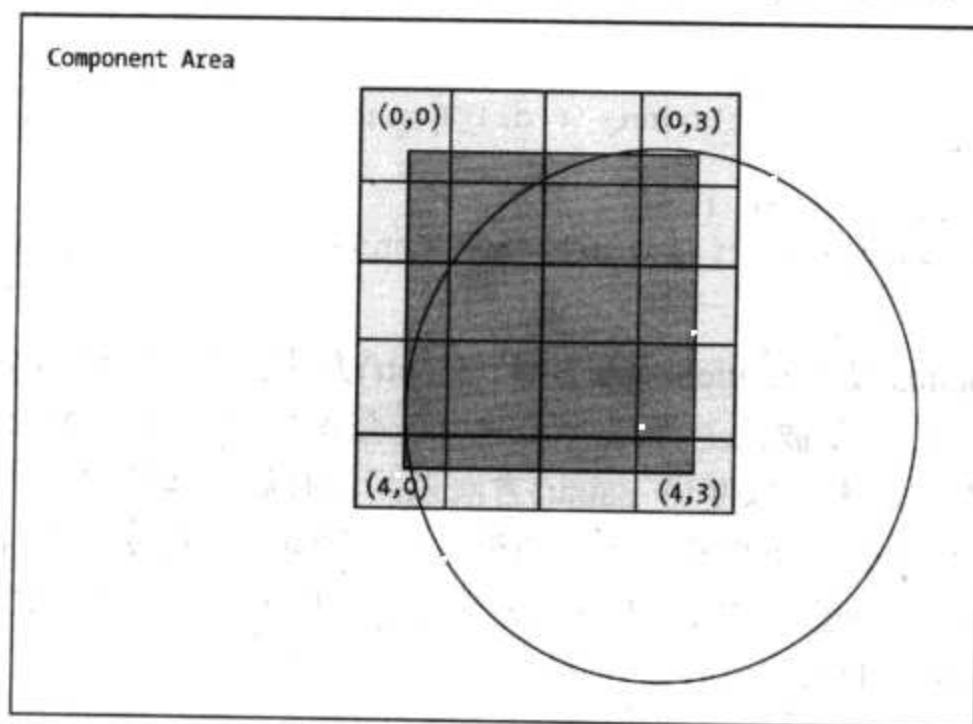


图 4-4 瓦片与调整的剪裁区域重叠

如 4-5 所示，注意，在经过调整的 clipBounds 以外但是仍然在 tileShape 剪裁区域以内的所有重叠的瓦片部分都会被绘制出来。这意味着瓦片的一部分可能会绘制在 originalClipBounds 的外面。一般来说，这不会成为问题，因为 originalClipBounds 只表示一个要被绘制的最小区域，超越它边界的任何额外绘制都是可以接受的。要实现很好的控制就需要计算 originalClipBounds 和实际 tileShape 的交集部分，而不是简单地使用包围 tileShape 的矩形。这对任意形状的 tileShape 实例而言，计算的代价可能会很大。如果在某一些环境中确实需要精确地控制动画，那么就需要考虑动态地调整 tileShape。

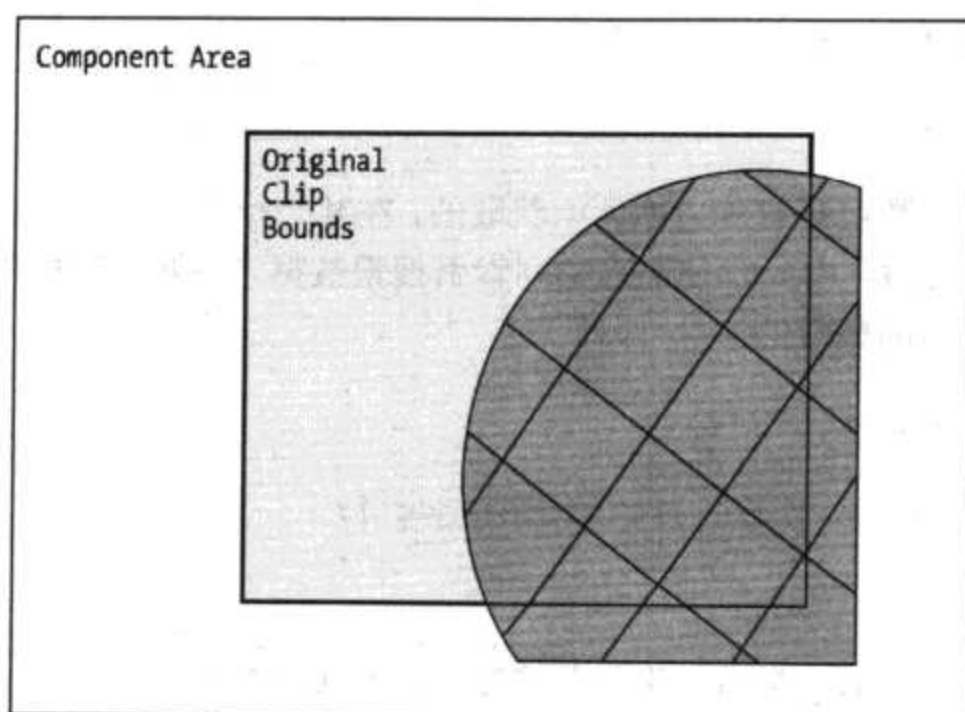


图 4-5 绘制在原始剪裁边界以外的一部分瓦片

```
int rows = getTileRows ( );

int columns = getTileColumns ( );

for ( int row = minRow; row <= maxRow; row++ )
{
    int r = row % rows;

    if ( r < 0 )
    {
        r += rows;
    }

    byte [ ] rowTileData = tileMap [ r ];
```

变量 `r` 是标准化以后的 `row` 值。例如在 `tileMap` 中只有两行，但是要绘制的区域由于重复和回绕的关系，却可能在 `minMrow(-2)` 到 `maxMrow(+3)` 之间进行扩展。在这种情况下，`r` 的值可能为 0、1、0、1、0、1。

```
for ( int column = minColumn; column <= maxColumn; column++ )
{
    int c = column % columns;

    if ( c < 0 )
    {
        c += columns;
    }
```

对列而言，也要执行一个相类似的标准化步骤。

```
tileIcons [ 0xFF & rowTileData [ c ] ].paintIcon (
    component,
    graphics,
    column * tileWidth + offsetX,
```

```

        row * tileHeight + offsetY);
    }
}

```

注意, rowTileData 数组是从外层循环中获取的, 在某一行内选定的特定列是在内层循环中选定的。这避免在内层循环中同时使用行和列索引搜索数据。还要注意掩码操作是怎样将字节值转换为 0~255 范围内的整数的。

```

    if ( tileShape != null )
    {
        graphics.setClip ( originalClipBounds );
    }
}

```

在 paint()方法返回以前, 最后一步就是恢复 originalClipBounds。

```

private static int floorDivision (
    int point,
    int zero,
    int length )
///////////////////////////////////////////////////////////////////
{
    if ( point >= zero )
    {
        return ( point - zero ) / length;
    }

    return ( point - zero + 1 ) / length - 1;
}

```

私有方法 floorDivision()用来将像素坐标转换为单元格的位置。这里的 point 是虚拟世界中像素的位置。zero 或基线是偏移量。length 是单元格的宽度和高度。例如, 如果 zero 是 10 个像素, length 是 40 个像素, 在 10~49 之间的任何 point 值都将会对应单元格索引位置 0。在 -30~+9 之间任意 point 值都会映射到索引位置 -1。

4.2 ComponentUpdater 实现

NullComponentUpdater 什么事情都不做。ArrayComponentUpdater 让多个 ComponentUpdater 实例看起来像是一个实例。只要鼠标移动到屏幕的边界, EdgeScrollUpdater 就会滚动视图。ComponentUpdater 的这些实现都可以在包 com.croftsoft.core.animation.updater 中找到。

4.2.1 NullComponentUpdater

NullComponentUpdater 是 ComponentUpdater 的一个空对象单态实现。调用它的 update()方法时, 它实际上不任何事情。

4.2.2 ArrayComponentUpdater

类 `ArrayComponentUpdater` 使多个 `ComponentUpdater` 实例看起来就像是一个实例一样。它的实现与 `ArrayComponentPainter` 类相似，不过，它是在 `ComponentUpdater` 数组中循环迭代而不是在 `ComponentPainter` 数组中。要想将一组不同的行为与一个精灵相关联时，这个类是非常有用的。

4.2.3 EdgeScrollUpdater

只要鼠标移动到屏幕的边界，`EdgeScrollUpdater` 就滚动屏幕视图。这对于虚拟世界的地图比屏幕一次能够显示的区域大的那些游戏来说是非常有用的。用户可以使用鼠标让视图按照用户的需要进行滑动，这样就可以对整个虚拟世界地图进行浏览。在 `CroftSoft Collection` 的 applet `Tile` 中对这个类有介绍。

```
package com.croftsoft.core.animation.updater;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.croftsoft.core.animation.ComponentUpdater;

public final class EdgeScrollUpdater
    implements ComponentUpdater
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    private final static int DEFAULT_SCROLL_RATE = 1;

    private final static int EDGE_SIZE_DIVIDER = 10;

    //

    private final int width;

    private final int height;

    private final Dimension edgeSize;

    private final Rectangle bounds;

    private final int scrollRate;

    private final boolean wrapAround;
```

用户将鼠标在屏幕左边边界处停留一段时间时，虚拟世界的视图就会一直向西滑动，直到虚拟世界的最西边界。在那一点上，视图要么完全停止下来，要么就会绕到虚拟世界的另一个边界(最东边界)，并继续滑动。为了能够确定什么时候达到该平面虚拟世界的终点，就要给出虚拟世界的 `width` 和 `height`(单位是像素)。注意，虚拟世界的大小经常要比视图大很多。

```
private final Dimension edgeSize;

private final Rectangle bounds;

private final int scrollRate;

private final boolean wrapAround;
```

`edgeSize` 决定屏幕边界的宽度(单位是像素)。 `Bounds` 存储组件的当前大小。 `scrollRate` 决定屏幕在各个方向上滑动的速度，度量的单位是每帧多少像素。布尔标志 `wrapAround` 表示到虚拟世界尽头时，视图是要滑动到虚拟世界的另一个边界还是停止滑动。

```
private Point mousePoint;
```

```
private int    translateX;
```

```
private int    translateY;
```

非 `final` 实例变量 `mousePoint` 记录鼠标的位置。整型变量 `translateX` 和 `translateY` 存储虚拟世界视图的当前平移。

```
public EdgeScrollUpdater (
    JComponent component,
    int         width,
    int         height,
    Dimension   edgeSize,
    int         scrollRate,
    boolean     wrapAround )
////////////////////////////////////
{
    this.width = width;

    this.height = height;

    this.edgeSize = edgeSize;

    if ( scrollRate < 1 )
    {
        throw new IllegalArgumentException ( "scrollRate < 1" );
    }

    this.scrollRate = scrollRate;

    this.wrapAround = wrapAround;

    bounds = new Rectangle ( );

    component.addMouseListener (
        new MouseMotionAdapter ( )
        {
            public void mouseMoved ( MouseEvent mouseEvent )
            {
                mousePoint = mouseEvent.getPoint ( );
            }
        } );
}
```

主构造函数将 `mousePoint` 和 `component`(可能是一个 `AnimatedComponent`)关联在一起，而后者提供虚拟世界的视图。当用户将鼠标指针滑过组件的时候，就会产生 `MouseEvent` 实例，

该实例用 `point` 存储当前鼠标位置的 `x` 和 `y` 坐标。`MouseMotionAdapter` 的匿名内部扩展类捕获这些鼠标事件，并存储 `mousePoint`，以便在游戏循环的更新阶段中进行处理。

```
public EdgeScrollUpdater (
    JComponent component,
    int width,
    int height )
    //////////////////////////////////////////
{
    this ( component, width, height, null, DEFAULT_SCROLL_RATE, false );
}
```

而构造函数委托给带有一个空 `edgeSize`、一个 `DEFAULT_SCROLL_RATE`(每帧一个像素) 和 `false`(不回绕)的主构造函数。

```
public int getTranslateX ( ) { return translateX; }

public int getTranslateY ( ) { return translateY; }
```

`accessor` 方法返回视图的当前平移，单位是像素。

```
public void update ( JComponent component )
    //////////////////////////////////////////
{
    if ( mousePoint == null )
    {
        return;
    }
}
```

如果前面的 `mousePoint` 不是在组件的边界上，并且自最后一次更新以后，还没有产生一个新的鼠标事件，边界滚动逻辑就不会执行。

```
component.getBounds ( bounds );

int x = mousePoint.x;

int y = mousePoint.y;

boolean onEdge = false;

int edgeWidth;

int edgeHeight;

if ( edgeSize == null )
{
    edgeWidth = bounds.width / EDGE_SIZE_DIVIDER;

    edgeHeight = bounds.height / EDGE_SIZE_DIVIDER;
}
else
{
```

```

    edgeWidth = edgeSize.width;

    edgeHeight = edgeSize.height;
}

```

如果 `edgeSize` 为空，就使用默认的 `edgeWidth` 和默认的 `edgeHeight`，默认的 `edgeWidth` 为组件宽度的十分之一，而默认的 `edgeHeight` 为组件高度的十分之一。注意，如果在游戏进行的过程中，组件的大小发生变化，那么组件的宽和高也会跟着变化。

```

if ( x < edgeWidth )
{
    if ( wrapAround || ( translateX < 0 ) )
    {
        onEdge = true;

        translateX += scrollRate;

        if ( translateX > 0 )
        {
            if ( wrapAround )
            {
                translateX -= width;
            }
            else
            {
                translateX = 0;
            }
        }
    }
}

```

如果鼠标是在屏幕的边界上，那么视图就会移动(前提是回绕模式被启用或视图还没有到达虚拟世界的边界)。如果可能进行滚动，那么视图将会在水平方向上移动多个像素，移动像素的多少是由 `scrollRate` 决定的。如果这种滚动使视图移动到虚拟世界的边界以外，在启用了回绕模式的时候，视图就会移动到虚拟世界的另一端。如果没有启用回绕模式，视图就停在虚拟世界里最边界的位置上。

```

else if ( x > bounds.width - edgeWidth )
{
    if ( wrapAround || ( translateX > -( width - bounds.width ) ) )
    {
        onEdge = true;

        translateX -= scrollRate;
    }
}
if ( translateX < -( width - bounds.width ) )
{
    if ( wrapAround )
    {
        translateX += width;
    }
}

```



```

    }
    else
    {
        translateX = -( width - bounds.width );
    }
}

[...]

```

如果鼠标在屏幕的右边界上，执行的操作与鼠标在左边界上执行的操作基本相似，只是屏幕滚动的方向相反。对向左滚动视图而言，translateX 边界为 0。然而，对向右滚动的视图而言，translateX 边界为虚拟世界的宽度减去视图的宽度。

修改 translateY 的代码与修改 translateX 的代码非常相似，惟一不同的就是使用的是虚拟世界和视图的高度而不是宽度。注意，当鼠标放到组件的角上的时候，视图就会沿着对角线的方向移动。

```

    if ( !onEdge )
    {
        mousePoint = null;
    }
    else
    {
        component.repaint ( );
    }
}

```

当鼠标没有越过屏幕的边界时，mousePoint 被置为空。这在下一次更新过程中，会防止 mousePoint 被再次检查。如果鼠标在边界上，由于视图已经被移动了，所以就会请求整个组件的重绘。

```

public void translate ( Graphics graphics )
///////////////////////////////////////////////////
{
    graphics.translate ( translateX, translateY );
}
public void translate ( Point point )
///////////////////////////////////////////////////
{
    point.x += translateX;

    point.y += translateY;
}

public void translateReverse ( Graphics graphics )
///////////////////////////////////////////////////
{
    graphics.translate ( -translateX, -translateY );
}

public void translateReverse ( Point point )

```

```

////////////////////////////////////
{
    point.x -= translateX;

    point.y -= translateY;
}

```

这些便利方法为世界和视图坐标之间的转换提供了通用的操作。例如，可以在游戏循环的绘制阶段中调用 `translate(Graphics)` 方法，通过对给定的 `graphics` 上下文进行转换来切换视图。可以使用 `translateReverse(Point)` 方法将鼠标坐标转换为世界坐标。

4.3 ComponentAnimator 实现

空的 `NullComponentAnimator` 不做任何事情。`TileAnimator` 驱动一个滑动的瓦片图案。`FrameRateAnimator` 采样并显示帧速率。`CursorAnimator` 在鼠标的位置上显示一个 `Icon`。这些类都包含在 `com.croftsoft.core.animation.animator` 包中。

4.3.1 NullComponentAnimator

`NullComponentAnimator` 是 `ComponentAnimator` 的一个空对象单态实现。调用它的 `update()` 方法或 `paint()` 方法的时候，它实际上不做任何事情。

4.3.2 TileAnimator

类 `TileAnimator` 驱动一个滑动的瓦片图案。它用 `TilePainter` 在绘制阶段绘制瓦片。在更新阶段，它修改 `TilePainter` 的偏移。

```

package com.croftsoft.core.animation.animator;

import java.awt.*;
import javax.swing.*;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.animation.painter.TilePainter;

public final class TileAnimator
    implements ComponentAnimator
{
    //////////////////////////////////////
    //////////////////////////////////////

    private final TilePainter tilePainter;

    private final int    deltaX;

    private final int    deltaY;

```


变量 `deltaX` 和 `deltaY` 分别存储水平和垂直两个方向的偏移率，单位是每帧多少个像素。

```
public TileAnimator (
    TilePainter tilePainter,
    int         deltaX,
    int         deltaY )
////////////////////////////////////
{
    NullPointerException.check ( this.tilePainter = tilePainter );

    this.deltaX = deltaX;

    this.deltaY = deltaY;

    if ( ( deltaX == 0 )
        && ( deltaY == 0 ) )
    {
        throw new IllegalArgumentException ( "deltaX and deltaY both 0" );
    }
}

public TileAnimator (
    int     offsetX,
    int     offsetY,
    Icon    icon,
    Shape   tileShape,
    int     deltaX,
    int     deltaY )
////////////////////////////////////
{
    this (
        new TilePainter ( offsetX, offsetY, icon, tileShape ),
        deltaX,
        deltaY );
}
```

构造函数存储或创建 `TilePainter` 和 `delta` 的值。

```
public void update ( JComponent component )
////////////////////////////////////
{
    int offsetX = tilePainter.getOffsetX ( );

    int offsetY = tilePainter.getOffsetY ( );

    int x = offsetX - deltaX;

    int y = offsetY - deltaY;
```

变量 `x` 和 `y` 存储更新后的 `offsetX` 和 `offsetY` 的值。

```
int tileWidth = tilePainter.getTileWidth ( );
```

```

int tileHeight = tilePainter.getTileHeight ( );

int tilesWide = tilePainter.getTileColumns ( );

int tilesHigh = tilePainter.getTileRows ( );

if ( x < 0 )
{
    x += ( tileWidth * tilesWide );
}
else if ( x >= tileWidth * tilesWide )
{
    x -= ( tileWidth * tilesWide );
}

if ( y < 0 )
{
    y += ( tilesHigh * tileHeight );
}
else if ( y >= tilesHigh * tileHeight )
{
    y -= ( tilesHigh * tileHeight );
}

```

将新的偏移值进行标准化，以使其保持在一定范围以内。

```

tilePainter.setOffsetX ( x );

tilePainter.setOffsetY ( y );

component.repaint ( );
}

```

更新偏移值，并且请求整个组件的重绘。

```

public void paint (
    JComponent component,
    Graphics2D graphics )
{
    //////////////////////////////////////
    tilePainter.paint ( component, graphics );
}

```

paint()方法只是简单地委托给 tilePainter。

如果需要连续地运行动画，像在 CroftSoft Collection 的 applet Sprite 中那样连续地移动砖块背景和云层，那就可以使用 TileAnimator 类。如果需要以一种定制的方式，动态地调整偏移的时候，使用 TilePainter 类。

4.3.3 FrameRateAnimator

类 FrameRateAnimator 采样并显示帧速率。帧速率就是在采样周期中帧的计数除以实际用

的时间。应用这个类的示例在 CroftSoft Collection 的 applet Mars 中，可以通过 F 键启用或关闭该功能。

```
package com.croftsoft.core.animation.animator;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.awt.font.FontLib;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.math.MathConstants;

public final class FrameRateAnimator
    implements ComponentAnimator
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    private static final long SAMPLE_PERIOD_IN_MILLIS
        = 10 * MathConstants.MILLISECONDS_PER_SECOND;

    private static final int MAX_FRAME_RATE = 999;

    这里的采样周期是 10 秒。FrameRateAnimator 最多可以显示三位数字的帧率，即 999fps。

    private final Color    color;

    private final Font     font;

    private final Rectangle repaintBounds;

    private final float    x;

    private final float    y;

    //

    private boolean disabled;

    private boolean oldDisabled;

    private long    frameCount;

    private long    lastUpdateTime;

    private double   frameRate;

    private String   frameRateString;

    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////
```

```

////////////////////////////////////
public FrameRateAnimator (
    Color          color,
    Font           font,
    Rectangle2D    textLayoutBounds )
////////////////////////////////////
{
    NullPointerException.check ( this.color = color );

    NullPointerException.check ( this.font = font );

    NullPointerException.check ( textLayoutBounds );

    x = ( float ) textLayoutBounds.getX ( );

    y = ( float ) -textLayoutBounds.getY ( );

    repaintBounds = textLayoutBounds.getBounds ( );

    repaintBounds.y = 0;

    frameRateString = "???";
}

public FrameRateAnimator (
    Component      component,
    Color          color )
////////////////////////////////////
{
    this (
        color,
        component.getFont ( ),
        FontLib.getTextLayoutBounds (
            component, Integer.toString ( MAX_FRAME_RATE ) ) );
}

public FrameRateAnimator ( Component component )
////////////////////////////////////
{
    this ( component, component.getForeground ( ) );
}

////////////////////////////////////
////////////////////////////////////

public void update ( JComponent component )
////////////////////////////////////
{
    if ( oldDisabled != disabled )
    {
        component.repaint ( );
    }
}

```



```

        oldDisabled = disabled;
    }

```

如果前一帧中没有启用 `FrameRateAnimator`，那么在当前帧中通过键盘就可启用它(反之亦然)。 `update()` 方法请求组件的重绘，以便显示或清除帧速率。

```

if ( disabled )
{
    return;
}

long updateTime = System.currentTimeMillis ( );

frameCount++;

```

每次调用 `update()` 方法时， `frameCount` 都会递增。

```

long timeDelta = updateTime - lastUpdateTime;

if ( timeDelta < SAMPLE_PERIOD_IN_MILLIS )
{
    return;
}

```

如果 10 秒的采样周期还没有到达， `update()` 方法不会继续更新显示。

```

frameRate = frameCount * MathConstants.MILLISECONDS_PER_SECOND
    / ( double ) timeDelta;

```

注意，必须首先将 `timeDelta` 的单位由毫秒转换为秒，然后才在这个计算中使用的。

```

if ( frameRate > MAX_FRAME_RATE )
{
    frameRateString = ">>>";
}
else
{
    frameRateString = Long.toString ( Math.round ( frameRate ) );
}

```

`frameRate` 被四舍五入为最接近的 `long` 值，然后再转换为 `String`。如果 `frameRate` 比三位数字还大，即大于 999，那么就显示 ">>>"。

```

frameCount = 0;
lastUpdateTime = updateTime;
component.repaint ( repaintBounds );
}

```

更新 `frameRateString` 的时候，就会在显示帧速率的地方，请求组件的重绘。同时还重置 `frameCount` 和 `lastUpdateTime`，开始另一个采样周期。

```

public void paint (

```

```

    JComponent component,
    Graphics2D graphics )
    //////////////////////////////////////
{
    if ( disabled )
    {
        return;
    }
    graphics.setColor ( color );

    graphics.setFont ( font );

    graphics.drawString ( frameRateString, x, y );
}

```

如果启用了帧速率显示，`paint()`方法就会在屏幕的左上角显示 `frameRateString` 的值。

[...]

```

public void toggle ()
    //////////////////////////////////////
{
    disabled = !disabled;
}

```

由于可以关闭或开启 `FrameRateAnimator`，所以在游戏中包含它不会带来什么危害。知道键盘组合键操作的玩家可以开启这个功能，从而知道动画在自己机器平台上运行速度的快慢。

4.3.4 CursorAnimator

`CursorAnimator` 在鼠标的位置上驱动一个 `Icon`。在移动鼠标指针的时候，`Icon` 也会跟着移动。当按下鼠标键的时候，`Icon` 会发生改变。该类的示例参见 `CroftSoft Collection` 中的 applet `Sprite`。

```

package com.croftsoft.core.animation.animator;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.croftsoft.core.animation.ComponentAnimator;

public class CursorAnimator
    implements ComponentAnimator, MouseListener, MouseMotionListener
    //////////////////////////////////////
    //////////////////////////////////////
{
    private final Rectangle oldPaintBounds;

    private final Rectangle newPaintBounds;

```


final 变量 `oldPaintBounds` 和 `newPaintBounds` 在更新阶段请求重绘。

```
private Icon    currentIcon;

private Icon    nextIcon;

private Icon    mouseReleasedIcon;

private Icon    mousePressedIcon;

private int     hotSpotX;

private int     hotSpotY;

private int     x;

private int     y;

private boolean updated;
```

`currentIcon` 是当前绘制出来的 `Icon`。`nextIcon` 是下一次更新以后要使用的 `Icon`。`mouseReleasedIcon` 是没有按下鼠标键时使用的 `Icon`，相反，`mousePressedIcon` 是按下鼠标键时使用的 `Icon`。`hotSpotX` 和 `hotSpotY` 是在鼠标指针位置上绘制 `currentIcon` 的偏移。`x` 和 `y` 的值是最新鼠标指针位置的坐标。布尔标志 `updated` 用来表示自上一次更新以后，是不是又发生了什么变化。

```
public CursorAnimator (
    Icon    mouseReleasedIcon,
    Icon    mousePressedIcon,
    Point    hotSpot,
    Component component )
////////////////////////////////////
{
    setMouseReleasedIcon ( mouseReleasedIcon, hotSpot );

    setMousePressedIcon ( mousePressedIcon , hotSpot );

    oldPaintBounds = new Rectangle ( );

    newPaintBounds = new Rectangle ( );

    if ( component != null )
    {
        component.addMouseListener ( this );

        component.addMouseListener ( this );

        component.setCursor ( new Cursor ( Cursor.CROSSHAIR_CURSOR ) );
    }
}
```

如果将一个非空的 component 作为参数传递给构造函数，那么就会将 CursorAnimator 添加为 MouseListener 和 MouseMotionListener，这样 CursorAnimator 可以管理鼠标事件。另外，组件的 Cursor 会被更改为 CROSSHAIR_CURSOR。

```
[...]

public void setMouseReleasedIcon (
    Icon    mouseReleasedIcon,
    Point   hotSpot )
//////////
{
    this.mouseReleasedIcon = mouseReleasedIcon;

    updateHotSpot ( hotSpot );
}

[...]
```

当 mousePressedIcon 或 mouseReleasedIcon 发生改变的时候，就会自动更新 hotSpot。

```
public void update ( JComponent component )
//////////
{
    if ( !updated )
    {
        return;
    }

    updated = false;
```

如果自上一次更新以来，没有发生任何变化，那么 update() 方法就不做任何事情，也不会请求重绘。

```
currentIcon = nextIcon;

if ( currentIcon != null )
{
    newPaintBounds.x = x;

    newPaintBounds.y = y;

    newPaintBounds.width = currentIcon.getIconWidth ( );

    newPaintBounds.height = currentIcon.getIconHeight ( );
}
else
{
    newPaintBounds.width = 0;
}
```

如果 currentIcon 已经发生了改变，或者已经移动了鼠标指针的位置，那么就必须更新 newPaintBounds。


```

    component.repaint ( oldPaintBounds );

    component.repaint ( newPaintBounds );

    oldPaintBounds.setBounds ( newPaintBounds );
}

```

update()方法最后请求鼠标原来位置和鼠标现在位置上的重绘。

```

public void paint (
    JComponent component,
    Graphics2D graphics )
///////////////////////////////////////////////////
{
    if ( currentIcon != null )
    {
        currentIcon.paintIcon ( component, graphics, x, y );
    }
}

```

当鼠标指针超出组件区域的时候，**currentIcon** 就为空。

```

public void mouseClicked ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
}

```

```

public void mouseEntered ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
    updated = true;

    nextIcon = mouseReleasedIcon;
}

```

```

public void mouseExited ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
    updated = true;

    nextIcon = null;
}

```

```

public void mousePressed ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
    updated = true;

    nextIcon = mousePressedIcon;
}

```

```

public void mouseReleased ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
}

```

```

{
    updated = true;

    nextIcon = mouseReleasedIcon;
}

```

这些 `MouseListener` 方法触发对 `nextIcon` 的更改。下一次更新以后，`nextIcon` 就变成了 `CurrentIcon`。

```

public void mouseDragged ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
    mouseMoved ( mouseEvent );
}

public void mouseMoved ( MouseEvent mouseEvent )
///////////////////////////////////////////////////
{
    if ( currentIcon == null )
    {
        return;
    }

    updated = true;

    x = mouseEvent.getX ( ) - hotSpotX;

    y = mouseEvent.getY ( ) - hotSpotY;
}

```

这些 `MouseMotionListener` 方法更改了 `Icon` 的位置，以使它能跟着鼠标指针移动。

```

private void updateHotSpot ( Point hotSpot )
///////////////////////////////////////////////////
{
    updated = true;

    if ( hotSpot != null )
    {
        hotSpotX = hotSpot.x;

        hotSpotY = hotSpot.y;
    }
    else if ( mouseReleasedIcon != null )
    {
        hotSpotX = mouseReleasedIcon.getIconWidth ( ) / 2;

        hotSpotY = mouseReleasedIcon.getIconHeight ( ) / 2;
    }
    else if ( mousePressedIcon != null )
    {
        hotSpotX = mousePressedIcon.getIconWidth ( ) / 2;
        hotSpotY = mousePressedIcon.getIconHeight ( ) / 2;
    }
}

```



```

    }
    else
    {
        hotSpotX = 0;

        hotSpotY = 0;
    }
}

```

私有方法 `updateHotSpot()` 设置 `hotSpotX` 和 `hotSpotY` 值。如果给的是一个空 `hotSpot` 参数, 那么热区(hot spot)就是该 `Icon`。

4.4 Sprite 实现

在这一节中, 将介绍一个接口 `ComponentAnimator` 的扩展——`Sprite`, 它描绘的是一个在虚拟空间中移动的对象。抽象类 `AbstractSprite` 提供了部分实现。类 `IconSprite` 通过使用一个 `Icon` 给出了完整实现。类 `BounceUpdater` 是 `ComponentUpdater` 的一个示例类, 这个类知道如何管理 `Sprite`。类 `IconSequenceUpdater` 通过一个 `Icon` 数组产生一个 `IconSprite` 序列。

4.4.1 Sprite

包 `com.croftsoft.core.animation` 中的接口 `Sprite` 扩展了 `ComponentAnimator`, 以提供在虚拟空间中移动游戏实体的一些辅助方法。它还提供了一些方法来确定是否发生了碰撞, 以及屏幕的哪一个区域需要进行重绘。

```

package com.croftsoft.core.animation;

import java.awt.Rectangle;
import java.awt.Shape;

public interface Sprite
    extends ComponentAnimator
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    public double getX      ( );

    public double getY      ( );

    public double getZ      ( );

```

`Sprite` 对虚拟空间中的坐标使用的变量类型是 `double` 而不是 `int`。`Sprite` 被假想成生活在分层的二维虚拟空间中。属性 `z` 表示 `Sprite` 的 `z` 坐标, 这样动画引擎就可以决定先绘制哪一个精灵。

```

    public double getHeading ( );

    public double getVelocity ( );

```

Sprite 以内部状态的形式维护它自己的 heading 和 velocity。一般而言, heading 是以弧度的形式给出的, 而 velocity 是以每秒多少米的形式给出的, 但是也不总是如此。例如, 有些具体的实现可能使用的就是每秒多少个像素。

```
public void getCollisionBounds ( Rectangle collisionBounds );

public Shape getCollisionShape ( );
```

Sprite 可以和虚拟世界中的其他实体发生碰撞。方法 getCollisionBounds() 被 Sprite 用来确定碰撞区域的 Rectangle 的范围。方法 getCollisionShape() 在返回实际的碰撞区域方面更加精确。注意, 内核包 java.awt 中的接口 Shape 和它的实现, 例如 Rectangle, 都提供了测试 Shape 是否与矩形相交或包含某一个点的方法。可以使用这些方法确定在虚拟空间中是否发生了碰撞, 或玩家是否使用鼠标单击了某一个具体的 Sprite。

```
public void getPaintBounds ( Rectangle paintBounds );
```

当 Sprite 在屏幕上移动的时候, paintBounds 确定需要重绘的区域。它可能与 collisionBounds 或 collisionShape 不完全相同。例如, 在等轴视图中, paintBounds 可能就会比 collisionBounds 高一些。另外, 即使 paintBounds 是一个矩形, 碰撞球的 Sprite 也可能会有一个圆形的 collisionShape。

```
public void setX      ( double x );

public void setY      ( double y );

public void setZ      ( double z );

public void setHeading ( double heading );

public void setVelocity ( double velocity );
```

通过设置属性 x 和 y 的值, 可以在虚拟空间中移动精灵。更改属性 z 的值可以确定该精灵是在前景上还是在背景上。还可以改变 Sprite 的 heading 和 velocity。

4.4.2 AbstractSprite

包 com.croftsoft.core.animation.sprite 中的抽象类 AbstractSprite 给出了接口 Sprite 的部分实现。

```
package com.croftsoft.core.animation.sprite;

import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentPainter;
import com.croftsoft.core.animation.ComponentUpdater;
import com.croftsoft.core.animation.Sprite;
import com.croftsoft.core.animation.painter.NullComponentPainter;
import com.croftsoft.core.animation.updater.NullComponentUpdater;
import com.croftsoft.core.lang.NullArgumentException;
```



```

public abstract class AbstractSprite
    implements Sprite
    //////////////////////////////////////
    //////////////////////////////////////
{

    protected double          x;

    protected double          y;

    protected double          z;

    protected double          heading;

    protected double          velocity;

    protected ComponentUpdater componentUpdater;

    protected ComponentPainter componentPainter;

```

由于 Sprite 扩展了接口 ComponentAnimator，所以 Sprite 的所有实现都必须提供 ComponentAnimator 的方法 update() 和 paint()。AbstractSprite 将这两个方法分别委托给 componentUpdater 和 componentPainter。

```

public AbstractSprite (
    double          x,
    double          y,
    double          z,
    double          heading,
    double          velocity,
    ComponentUpdater componentUpdater,
    ComponentPainter componentPainter )
    //////////////////////////////////////
{
    this.x = x;

    this.y = y;

    this.z = z;

    this.heading = heading;

    this.velocity = velocity;

    setComponentUpdater ( componentUpdater );

    setComponentPainter ( componentPainter );
}

public AbstractSprite ( )
    //////////////////////////////////////

```

```
{
    this ( 0.0, 0.0, 0.0, 0.0, 0.0, null, null );
}
```

主构造函数只是以实例变量的形式存储参数。没有参数的便利构造函数只是向主构造函数的参数传递 0 值或空值。

```
public double getX ( ) { return x; }

[...]
public void getCollisionBounds ( Rectangle collisionBounds )
///////////////////////////////////////////////////////////////////
{
    collisionBounds.setBounds ( getCollisionShape ( ).getBounds ( ) );
}
```

大部分 accessor 方法都只是返回以实例变量的形式存储的值。注意，在 `AbstractSprite` 中，并没有提供 `getCollisionShape()` 方法和 `getPaintBounds()` 方法的默认实现。方法 `getCollisionBounds()` 使用接口 `Sprite` 的 `getCollisionShape()` 方法和接口 `Shape` 的 `getBounds()` 方法来获取 `collisionBounds`。在实现 `getCollisionShape()` 的时候，为了能够提高性能，可以重写 `getCollisionBounds()` 方法的这种默认实现。

```
public void setX ( double x ) { this.x = x; }

[...]

public void setComponentUpdater (
    ComponentUpdater componentUpdater )
///////////////////////////////////////////////////////////////////
{
    if ( componentUpdater == null )
    {
        componentUpdater = NullComponentUpdater.INSTANCE;
    }

    this.componentUpdater = componentUpdater;
}

[...]
```

大部分 mutator 方法都只是以实例变量的形式存储值。如果参数为空的话，方法 `setComponentUpdater()` 和方法 `setComponentPainter()` 就存储单态的空对象的引用。

```
public void update ( JComponent component )
///////////////////////////////////////////////////////////////////
{
    componentUpdater.update ( component );
}

public void paint (
    JComponent component,
```



```

        Graphics2D graphics )
    //////////////////////////////////////
    {
        componentPainter.paint ( component, graphics );
    }

```

只是将方法 `update()` 和 `paint()` 分别委托给 `componentUpdater` 和 `componentPainter`。也可以在子类中重写这些方法。

4.4.3 IconSprite

类 `IconSprite` 是 `Sprite` 类的一个实现，支持它的是 `Icon` 类的一个实例。

```

package com.croftsoft.core.animation.sprite;

import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.Shape;
import javax.swing.Icon;
import javax.swing.JComponent;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.animation.ComponentUpdater;
import com.croftsoft.core.animation.painter.NullComponentPainter;
import com.croftsoft.core.animation.updater.NullComponentUpdater;

public class IconSprite
    extends AbstractSprite
    //////////////////////////////////////
    //////////////////////////////////////
    {

    private final Rectangle paintBounds;

    //

    private Icon icon;

```

类 `IconSprite` 扩展了 `AbstractSprite`。它将它的 `paintBounds` 作为可变的 `final` 变量并将 `icon` 作为非 `final` 变量，以便它可以进行更改。

```

public IconSprite (
    double          x,
    double          y,
    double          z,
    double          heading,
    double          velocity,
    ComponentUpdater componentUpdater,
    Icon            icon )
    //////////////////////////////////////
    {
        super ( x, y, z, heading, velocity, componentUpdater,

```

```

        NullComponentPainter.INSTANCE );

    paintBounds = new Rectangle ( );

    setX ( x );

    setY ( y );

    setIcon ( icon );
}

```

由于 `IconSprite` 重写了其父类 `AbstractSprite` 的 `paint()` 方法，所有也就不再需要 `componentPainter` 的委托了。`NullComponentPainter` 实例是以空对象形式提供给父类构造函数的。由于方法 `setX()`、`setY()`和 `setIcon()`更新了 `paintBounds`，所以它们必须在构造函数中将 `paintBounds` 对象初始化以后再进行调用。

```

public IconSprite ( Icon icon )
///////////////////////////////////////////////////////////////////
{
    this ( 0.0, 0.0, 0.0, 0.0, 0.0,
          NullComponentUpdater.INSTANCE, icon );
}

```

这个便利构造函数将所有的值都初始化为0或空对象(除icon以外)。

```

public Icon getIcon ( ) { return icon; }
public Shape getCollisionShape ( )
///////////////////////////////////////////////////////////////////
{
    return paintBounds;
}

public void getCollisionBounds ( Rectangle collisionBounds )
///////////////////////////////////////////////////////////////////
{
    collisionBounds.setBounds ( paintBounds );
}

public void getPaintBounds ( Rectangle paintBounds )
///////////////////////////////////////////////////////////////////
{
    paintBounds.setBounds ( this.paintBounds );
}

```

`IconSprite`也为`collisionBounds`和`collisionShape`使用了`paintBounds`。

```

public void setX ( double x )
///////////////////////////////////////////////////////////////////
{
    super.setX ( x );

    paintBounds.x = ( int ) x;
}

```



```

}

public void setY ( double y )
////////////////////////////////////////////////////
{
    super.setY ( y );

    paintBounds.y = ( int ) y;
}

```

当 `IconSprite` 在虚拟空间中的位置被更新的时候，也同时更新 `paintBounds`。

```

public void setIcon ( Icon icon )
////////////////////////////////////////////////////
{
    NullPointerException.check ( this.icon = icon );

    paintBounds.width = icon.getIconWidth ( );

    paintBounds.height = icon.getIconHeight ( );
}

```

当 `icon` 变化的时候，也同时更新 `paintBounds` 的 `width` 和 `height`。

```

public void paint (
    JComponent component,
    Graphics2D graphics )
////////////////////////////////////////////////////
{
    icon.paintIcon ( component, graphics, ( int ) x, ( int ) y );
}

```

`IconSprite` 的 `paint()` 方法提供了 `icon` 带有 `x` 和 `y` 值的 `paintIcon()` 方法。这假定虚拟空间中的坐标的单位是像素，或是一比一映射到像素上。

4.4.4 BounceUpdater

为了使 `AbstractSprite` 的子类(例如 `IconSprite`)能执行一些任务，需要在子类中重写它的 `update()` 方法，或提供 `ComponentUpdater` 的一个委托实现。包 `com.croftsoft.core.animation.update` 中的类 `BounceUpdater` 是一个 `ComponentUpdater` 实现，该实现知道如何管理一个 `Sprite`。它使一个 `Sprite` 弹离 `Rectangle` 的墙面。在 `CroftSoft Collection` 的 applet `Sprite` 中使用了 `BounceUpdater`。

```

package com.croftsoft.core.animation.updater;

import java.awt.Rectangle;
import java.awt.Shape;
import javax.swing.JComponent;

import com.croftsoft.core.animation.Clock;
import com.croftsoft.core.animation.ComponentUpdater;

```

```
import com.croftsoft.core.animation.Sprite;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.math.MathConstants;
```

类 `BounceUpdater` 导入了 `com.croftsoft.core.animation` 包中的接口 `Clock`。`Clock` 实现提供了当前的时间，单位是纳秒。可以在 `om.croftsoft.core.animation.clock` 包中找到接口 `Clock` 的实现。

```
public final class BounceUpdater
    implements ComponentUpdater
    //////////////////////////////////////
    //////////////////////////////////////
{

    private final Sprite      sprite;

    private final Rectangle   bounds;

    private final Clock       clock;

    private final Rectangle   collisionBounds;

    private final Rectangle   newPaintBounds;

    private final Rectangle   oldPaintBounds;
```

精灵在矩形 `bounds` 内弹跳。`clock` 用来确定在一次更新中精灵要移动多远。使用 `collisionBounds` 来确定 `sprite` 是否已经碰到了墙。请求的重绘区域将是 `sprite` 的 `oldPaintBounds` 和 `newPaintBounds` 的联合区域。

```
private long lastUpdateTimeNanos;
```

`lastUpdateTimeNanos` 记录上次更新的时间，单位是纳秒。

```
public BounceUpdater (
    Sprite      sprite,
    Rectangle   bounds,
    Clock       clock )
    //////////////////////////////////////
{
    NullArgumentException.check ( this.sprite = sprite );

    NullArgumentException.check ( this.bounds = bounds );

    NullArgumentException.check ( this.clock = clock );

    collisionBounds = new Rectangle ( );

    newPaintBounds  = new Rectangle ( );

    oldPaintBounds  = new Rectangle ( );
}
```


final 实例变量 `collisionBounds`、`newPaintBounds` 和 `oldPaintBounds` 在构造函数中创建一次，我们可以重复使用它们存储新值，因为它们是可变的。

```
public void update ( JComponent component )
///////////////////////////////////////////////////
{
    long updateTimeNanos = clock.currentTimeNanos ( );

    if ( updateTimeNanos == lastUpdateTimeNanos )
    {
        return;
    }
}
```

如果使用的是低精度的 `clock`，那么当前的 `updateTimeNanos` 可能会与 `lastUpdateTimeNanos` 相同，即使自上一次调用 `update()` 以后，已经明显地消耗了一些时间。在这种情况下，也可能就不进行更新。

```
double timeDelta
    = ( updateTimeNanos - lastUpdateTimeNanos )
    / ( double ) MathConstants.NANOSECONDS_PER_SECOND;
```

```
lastUpdateTimeNanos = updateTimeNanos;
```

在 `lastUpdateTimeNanos` 更新以前，它用来确定自上一次更新以后的 `timeDelta`，单位是秒。

```
double y = sprite.getY ( );

double heading = sprite.getHeading ( );

double velocity = sprite.getVelocity ( );

double delta_x = Math.cos ( heading ) * velocity * timeDelta;
double delta_y = Math.sin ( heading ) * velocity * timeDelta;

x += delta_x;

y += delta_y;
```

更新后的坐标是基于 `heading`、`velocity` 和 `timeDelta` 计算的。

```
int minX = bounds.x;

int minY = bounds.y;

sprite.getCollisionBounds ( collisionBounds );

int maxX = bounds.x + bounds.width - collisionBounds.width;

int maxY = bounds.y + bounds.height - collisionBounds.height;
```

`x` 和 `y` 的值是 `sprite` 左上角的坐标。为了防止精灵与边框的右下边界重叠，可以分别用 `sprite` 的宽和高来调整 `maxX` 和 `maxY` 的值。

```

boolean headingAlreadyAdjusted = false;

if ( x < minX )
{
    x = minX;

    if ( delta_x < 0.0 )
    {

        if ( heading > Math.PI )
        {
            heading = 3.0 * Math.PI - heading;
        }
        else
        {
            heading = Math.PI - heading;
        }

        headingAlreadyAdjusted = true;
    }
}

```

如果更新后的 `x` 坐标比 `minX` 还小，`sprite` 的位置就会处在 `bounds` 之外，因此必须将 `x` 的值设为 `minX`，以将它移到边框以内。如果向左移动，`sprite` 移到边框以外，就会碰上左边的墙壁。这种弹跳导致它的 `heading` 发生变化——这种变化的方式与镜面反射的方式一样。

```

else if ( x > maxX )
{
    [...]
}

```

`sprite` 从右边墙壁回弹的代码与此相似。

```

if ( y < minY )
{
    y = minY;

    if ( delta_y < 0.0 )
    {
        if ( !headingAlreadyAdjusted )
        {
            heading = 2.0 * Math.PI - heading;
        }
    }
}

```

这段代码让 `sprite` 从顶部边界回弹。如果检查 `x` 值的时候，`heading` 被调整过了，这里就不再对它作其他调整。这可以防止 `y` 值检查恢复刚刚在 `x` 值检查中作的 `heading` 更改。

```

else if ( y > maxY )
{
    [...]
}

```


从下边界回弹sprite的代码与上面的代码相类似。

```
sprite.getPaintBounds ( oldPaintBounds );
```

```
sprite.setX ( x );
```

```
sprite.setY ( y );
```

在更新sprite坐标以前，首先保存oldPaintBounds的值。

```
sprite.setHeading ( heading );
```

```
sprite.getPaintBounds ( newPaintBounds );
```

```
oldPaintBounds.add ( newPaintBounds );
```

```
component.repaint ( oldPaintBounds );
```

```
}
```

请求重绘的区域是 oldPaintBounds 和 newPaintBounds 的联合区域。这是假定原位置和新位置相互接近到原 sprite 和新 sprite 的区域能够重叠在一起的情况。

4.4.5 IconSequenceUpdater

com.croftsoft.core.animation.updater 包中的 IconSequenceUpdater 操纵一个 IconSprite，如图 4-6 所示。它通过一个 Icon 数组，使 IconSprite 形成一个序列。在 CroftSoft Collection 的 applet Zombie 中，Icon 数组包含一个蛇神(zombie)的 ImageIcon，该蛇神的左脚向前，还包含另外一个蛇神的 ImageIcon，这个蛇神的右脚向前。通过每秒交替使用这两个图片，就可以显示出该蛇神的走动状态。

```
package com.croftsoft.core.animation.updater;
```

```
import java.awt.Rectangle;
```

```
import javax.swing.Icon;
```

```
import javax.swing.JComponent;
```

```
import com.croftsoft.core.lang.NullArgumentException;
```

```
import com.croftsoft.core.animation.Clock;
```

```
import com.croftsoft.core.animation.ComponentUpdater;
```

```
import com.croftsoft.core.animation.sprite.IconSprite;
```

```
public final class IconSequenceUpdater
```

```
    implements ComponentUpdater
```

```
    //////////////////////////////////////  
    //////////////////////////////////////
```

```
{
```

```
    private final IconSprite iconSprite;
```

```
    private final Icon [ ] icons;
```

```
    private final long framePeriodNanos;
```

```
private final Clock clock;

private final Rectangle oldPaintBounds;

private final Rectangle newPaintBounds;
```

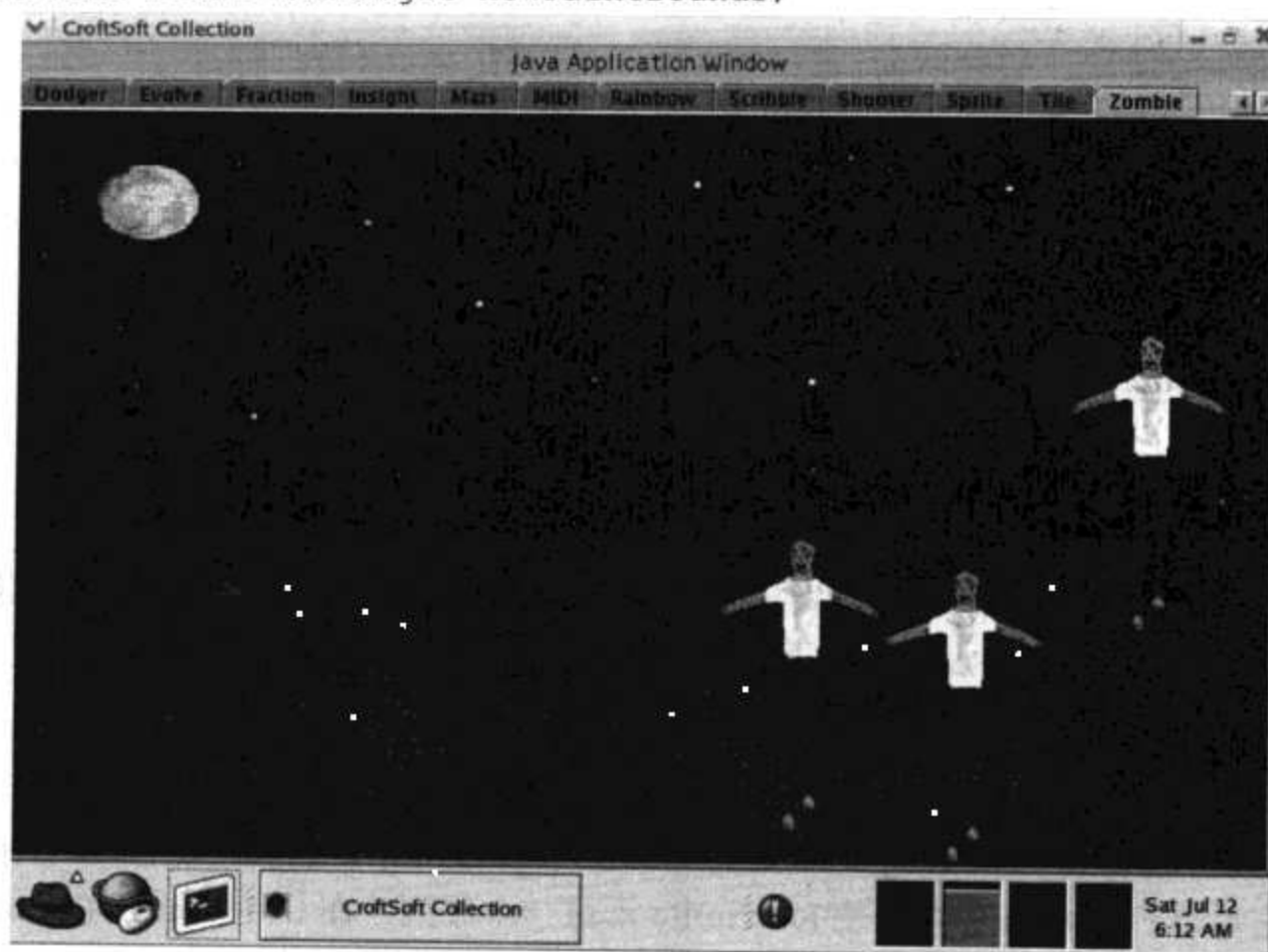


图 4-6 IconSpriteUpdater 示例

通过使用 icons 数组，并在每一个 Icon 之间使用 clock 添加一个以纳秒为单位的 framePeriodNanos 延迟，IconSprite 就会形成序列。请求重绘的区域将是 iconSprite 的 oldPaintBounds 和 newPaintBounds 的联合区域。

```
private long lastUpdateTimeNanos;
```

```
private int index;
```

lastUpdateTimeNanos 记录上次的更新时间，单位是纳秒。Index 指向 icons 数组中的下一个 Icon。

```
public IconSequenceUpdater (
    IconSprite iconSprite,
    Icon [ ]   icons,
    long       framePeriodNanos,
    Clock      clock )
{
    ////////////////////////////////////////////
    NullPointerException.check ( this.iconSprite = iconSprite );

    NullPointerException.check ( this.icons = icons );
```



```

this.framePeriodNanos = framePeriodNanos;

NullPointerException.check ( this.clock = clock );

oldPaintBounds = new Rectangle ( );

newPaintBounds = new Rectangle ( );
}

```

构造函数仅仅存储或初始化 **final** 类型的变量。

```

public void update ( JComponent component )
///////////////////////////////////////////////////////////////////
{
    long updateTimeNanos = clock.currentTimeNanos ( );

    if ( updateTimeNanos < lastUpdateTimeNanos + framePeriodNanos )
    {
        return;
    }

    lastUpdateTimeNanos = updateTimeNanos;

```

如果自 **Icon** 被更新以后,还没有到达 **framePeriodNanos**(单位是纳秒)延迟时间,那么 **update()** 方法就不做任何事情。

```

    iconSprite.getPaintBounds ( oldPaintBounds );

    iconSprite.setIcon ( icons [ index ] );

    iconSprite.getPaintBounds ( newPaintBounds );

    newPaintBounds.add ( oldPaintBounds );

    component.repaint ( newPaintBounds );

    index = ( index + 1 ) % icons.length;
}

```

由于 **icons** 数组中每一个 **Icon** 的大小可能都不一样,所以选用下一个 **Icon** 的时候, **oldPaintBounds** 可能与 **newPaintBounds** 不相等。然而,我们希望它们能够重叠,这两个区域的联合将要用于重绘请求。

4.5 小结

本章介绍了多个 **ComponentUpdater**、**ComponentPainter**、**ComponentAnimator** 和 **Sprite** 的实现,这些实现与前面几章中介绍的核心动画引擎类都是一致的。这些实现都是为了解释怎样实现您游戏中特定的类和构造自己的可重用动画库。

第 5 章 高级图形技术

如果您想得到安逸的话，那就充分利用您的时间。

—— 本杰明·富兰克林

这一章主要介绍多种高级图形方面的技术，包括硬件加速图像、多缓冲技术和全屏独占模式。硬件加速图像通过直接在显示存储器中存储和操作图像数据，可以用来提高帧速率。多缓冲技术使用附加图像缓冲区，使屏幕图像的产生可以在一个与事件分派线程独立的线程中完成。使游戏以全屏独占模式运行，它就可以填充整个屏幕；使游戏以全屏独占模式运行还需要设置显示模式，并需要防止 tearing(锯齿) (如果在刷新显示器的同时，也刷新了显示内存，这时偶尔就会出现一些小的波纹，这种小波纹就叫锯齿)。

5.1 硬件加速图像

硬件加速图像技术通过直接在显示内存中存储和操作图像数据，可以提高动画的帧速率。硬件加速的效果是随着机器变化的，因此，游戏必须能够在没有硬件加速的机器上也能以玩家可以接受的执行性能运行。内核包 java.awt 中的类 Image 和它的子类都提供了一个抽象层，这样游戏代码就总会连续地运行，而与某给定平台上是否有硬件加速功能无关。

5.1.1 Image

抽象类 Image 保持图像元素(像素)数据。由于这个类是抽象类，所以不能直接使用构造函数对它进行实例化。下面的代码介绍了使用内核包 javax.swing 中 ImageIcon 类的一个实例，从 JAR 文件中加载一个 Image 的简单技术。

```
URL imageURL = classLoader.getResource ( imageFilename );

if ( imageURL == null )
{
    throw new IllegalArgumentException (
        "unknown imageFilename: " + imageFilename );
}

ImageIcon imageIcon = new ImageIcon ( imageURL );

Image image = imageIcon.getImage ( );
```

Image 的这两个子类分别是 BufferedImage 和 VolatileImage。

5.1.2 BufferedImage

内核包 `java.awt.image` 中的类 `BufferedImage` 提供了直接访问和修改像素数据的方法。这说明可以获取 `Image` 内某一个给定的行和列位置上像素的颜色，还可以更改这个颜色。

可以在内存中通过调用 `BufferedImage` 的一个构造函数，直接创建 `BufferedImage` 的一个空实例。`BufferedImage` 的像素数据存储在系统常规内存中，而不是存储在带有硬件加速的显示内存中。这说明，在修改 `BufferedImage` 像素数据的时候，就不能利用任何硬件加速的绘制操作，而这种硬件加速的绘制操作在当前的机器平台基本上都可用。此外，每次显示 `BufferedImage` 的时候，像素数据必须从系统内存复制到显示内存，而这个过程可能是很慢的。将数据从内存的一个地方复制到另一个地方的这个过程被称为块转移(block line transfer)或简称为 blit。

可以使用下面给出的 `com.croftsoft.core.awt.image` 包中 `ImageLib` 类的静态方法，从 JAR 文件中加载 `BufferedImage`。

```
public static BufferedImage loadBufferedImage (
    String      imageFilename,
    ClassLoader classLoader )
    throws IOException
    //////////////////////////////////////
{
    InputStream inputStream
        = classLoader.getResourceAsStream ( imageFilename );

    if ( inputStream == null )
    {
        return null;
    }
    BufferedInputStream bufferedInputStream
        = new BufferedInputStream ( inputStream );

    BufferedImage bufferedImage = ImageIO.read ( bufferedInputStream );

    bufferedInputStream.close ( );

    return bufferedImage;
}
```

注意，当 `ImageLib` 类中的 `loadBufferedImage()` 方法返回 `BufferedImage` 的一个实例的时候，`ImageIcon` 类中的 `getImage()` 方法会返回 `Image` 的一个实例。一旦将 `BufferedImage` 加载到内存，就可以直接操作像素数据了。

5.1.3 VolatileImage

内核包 `java.awt.image` 中的类 `VolatileImage` 没有处理像素数据的 `accessor` 方法和 `mutator` 方法，但是它确实是在硬件加速的显示内存中存储像素数据的(如果该平台上硬件加速的显示存储器可用的话)。显示 `VolatileImage` 可能会非常快，因为数据可能已经在显卡的显存中了。

对 Java 游戏编程来说，近些年来 Java 编程语言最有意义的增强可能就是在核心 API 的 1.4 版中引入了 `VolatileImage` 类。在一个装有很好显卡的机器上，即使是每一帧都刷新整个屏幕，

现在也可以获得令人惊叹的帧速率。在这种假定下，在更新阶段请求整个组件区域的重绘，而不是请求对组件上需要重绘的那些区域进行重绘，可能要简单一些。这节省了计算哪些区域需要重绘所花费的时间，但是这样也有缺点，缺点就是在那些没有 `VolatileImage` 的硬件加速性能优势的平台上，动画显示的性能就会剧烈下降。在摩尔规律(Moore's Law)呈现之前的很长的一段时间内，我还是建议您应该继续努力，将每帧中需要重绘像素的平均数降到最小。

使用 `VolatileImage` 的一个缺点就是像素数据是极其不稳定的，例如，在有些平台上它可能总是要耗尽全部资源。当另外有一个进程，例如屏幕保护程序，决定重写显示内存的时候，就可能发生这种问题。直接使用 `VolatileImage` 实例需要使用它的 `contentsLost()` 方法来确定是否需要恢复图像数据。

我已在包 `com.croftsoft.core.animation.icon` 中创建了两个类，这两个类自动执行这些检查和恢复(check-and-restore)操作。这两个类都是 `Icon` 的实现，二者都在每次调用 `paintIcon()` 方法的时候，执行检查和恢复操作。类 `VolatileImageIcon` 使用一个规则且不变的 `Image` (该 `Image` 是以实例变量的形式存储的)来恢复 `VolatileImage` 的内容。当为了保留系统内存而将图像丢失的时候，类 `ResourceImageIcon` 就通过从 JAR 文件中重新加载数据的方式来恢复图像的内容。

使用 `VolatileImage` 的另外一个缺点就是它通常不支持半透明的颜色。这对精灵来说就成为一个问题，因为它们经常就处在半透明背景中。正是由于这个原因，我不经常使用 `VolatileImageIcon` 和 `ResourceImageIcon`。相反，我习惯使用下一节要介绍的那种自动图像技术。

5.1.4 自动图像

自动图像(automatic image)这个术语是由 Sun Microsystems 的工程师和视频游戏及电子娱乐(Video Game and Entertainment Electronics, VGEE)专家 Jeffrey Kesselman 在他 2001 年的论文 *Understanding the AWT Image Types* 中首次提出的。这篇文章介绍了一个图像类型，这个图像类型依赖 `BufferedImage` 和 `VolatileImage` 这二者的内部实例，将这二者的优点结合在一起。需要修改像素数据的时候，就在 `BufferedImage` 的内存区域对其进行修改，然后再将其复制给 `VolatileImage`，而后者的作用实际上就是一个高速缓存。当需要在屏幕上显示像素数据的时候，就使用加速的 `VolatileImage` 显示内存。

如果 `VolatileImage` 内存的内容已经丢失，自动图像将会检查到这个条件，并从系统内存的 `BufferedImage` 中恢复像素数据。再使用那篇文章中描述的一种所谓的“内部技巧”将透明的颜色信息成功地从 `BufferedImage` 转移到 `VolatileImage`。

这两个特性使自动图像技术和常规 `Image` 技术之间不存在任何区别，只是性能上有所提高。自 Java 1.4 版后，自动图像类型就是调用 `Component` 类的方法 `createImage()` 返回的默认图像类型。`createImage()` 方法是在 Java 1.0 版中引进的，因此很多以前依赖于它的 Java 游戏可能会突然感到帧速率性能上有一个很大的提高。

5.1.5 兼容图像

`createImage()` 方法返回 `Image` 的一个实例，这个实例不允许直接访问和修改像素数据。为了创建可以使用 `BufferedImage` 进行操作的自动图像类型，就需要使用内核包 `java.awt` 的类 `GraphicsConfiguration` 的 `createCompatibleImage()` 方法。

所谓兼容图像(compatible image)就是在一个 Image 中, 像素已经是 GraphicsConfiguration 使用的那种格式了。这使显示的速度得到提高, 因为每次将 Image 显示在屏幕上的时候, 就不再需要将像素数据转换成所要求的那种格式了。

在调用 createCompatibleImage()方法的时候, 可以以参数的形式指定使用什么类型的透明模式。透明度这个整型值是在内核包 java.awt 的类 Transparency 中定义的。没有透明像素的图像(例如背景瓦片)使用整型值 OPAQUE。包含完全透明像素加上完全不透明像素的图像(例如带有透明背景的精灵)使用 BITMASK。半透明的图像使用 TRANSLUCENT。

对现在的实现而言, 使用 TRANSLUCENT 图像对性能的影响将会非常大, 因为这种半透明的图像是没有经过加速的。可以在 CroftSoft Collection 的 Sprite 程序中, 通过启用 Paint fog and night 选项来观察它的影响。在启用这个选项的时候, 帧速率会有非常明显的下降, 因为雾和夜晚的这种效果是使用半透明的颜色实现的。一般而言, 几乎总是需要为图像使用 OPAQUE 和 BITMASK 模式。

另外一个需要考虑的性能问题是缩放比例的问题。如果对 Graphics 类的 drawImage()方法提供了宽度和高度参数, 在每次绘制图像的时候, 图像数据就都会缩放到给出的那个高和宽。在有些平台上使用缩放功能的时候, 就会失去 VolatileImage 的硬件加速优势。这个问题的解决办法是只要从 JAR 文件中加载这些图像, 就预先将图像缩放到理想的大小。

下面 ImageLib 类的静态便利方法 loadAutomaticImage()的功能就是从 JAR 文件中加载一个 BufferedImage, 创建一个理想大小的空白兼容自动图像, 再将像素数据从 BufferedImage 复制到自动图像中, 并在这个过程中转换和缩放图像数据。

```
public static BufferedImage loadAutomaticImage (
    String      imageFilename,
    int         transparency,
    Component    component,
    ClassLoader classLoader,
    Dimension   dimension )
    throws IOException
    //////////////////////////////////////
{
    NullPointerException.check ( imageFilename );

    NullPointerException.check ( component );

    if ( classLoader == null )
    {
        classLoader = component.getClass ( ).getClassLoader ( );
    }

    BufferedImage bufferedImage
        = loadBufferedImage ( imageFilename, classLoader );

    GraphicsConfiguration graphicsConfiguration
        = component.getGraphicsConfiguration ( );

    if ( graphicsConfiguration == null )
    {
```

```
        throw new IllegalStateException ( "null graphicsConfiguration" );
    }

    int width, height;

    if ( dimension == null )
    {
        width = bufferedImage.getWidth ( );

        height = bufferedImage.getHeight ( );
    }
    else
    {
        width = dimension.width;

        height = dimension.height;

        if ( width < 1 )
        {
            throw new IllegalArgumentException (
                "dimension.width < 1: " + width );
        }

        if ( height < 1 )
        {
            throw new IllegalArgumentException (
                "dimension.height < 1: " + height );
        }
    }

    BufferedImage automaticImage
        = graphicsConfiguration.createCompatibleImage (
            width, height, transparency );

    Graphics graphics = automaticImage.getGraphics ( );

    if ( dimension == null )
    {
        graphics.drawImage ( bufferedImage, 0, 0, null );
    }
    else
    {
        graphics.drawImage ( bufferedImage, 0, 0, width, height, null );
    }

    graphics.dispose ( );
    bufferedImage.flush ( );
    return automaticImage;
}
```


参数 `component` 是获取 `graphicsConfiguration` 所必需的，只有这样才能创建一个兼容图像。如果这个方法的 `classLoader` 参数为空，就会使用 `component` 的 `ClassLoader`。如果 `dimension` 参数为空，就会使用图像的原始大小而不会进行任何缩放。

注意，如果 `getGraphicsConfiguration()` 方法的返回值为空，那么 `loadAutomaticImage()` 就会抛出 `IllegalStateException` 异常。如果还没有将 `GraphicsConfiguration` 和 `component` 进行关联，就会发生这种情况。通过将 `loadAutomaticImage()` 的调用延迟到组件已经添加到 `Container` 以后，就能够避免这种问题的发生。一般来说，在游戏 applet 的 `init()` 方法内部调用 `loadAutomaticImage()` 都是安全的，但是在构造函数中调用它可能就会显得有些过早。

5.1.6 缓存算法

在我看来，使用自动图像总是要优于直接创建和操作 `VolatileImage` 实例，例如使用 `GraphicsConfiguration` 类的 `createCompatibleVolatileImage()` 方法或 `Component` 类的 `createVolatileImage()` 方法。自动图像提供了内置的逻辑，以便当存储在显存中的内容由于处理像屏幕保护程序这样的进程而丢失的时候，能够恢复内部的 `VolatileImage`。自动图像也考虑到处理 `VolatileImage` 带来的半透明问题，这样用户就可以对带有透明背景的精灵使用硬件加速。但是，选择直接使用 `VolatileImage` 而不是自动图像的惟一原因可能就是使用自动图像缓存算法有困难。

缓存算法就是由于不同的使用方法，而决定自动图像有着不同性能的一种算法。前面已经提过，缓存算法表现得相当灵巧。如果频繁地在自动图像里修改像素数据，也就是每一帧都要修改一次，那么缓存成功地命中 `VolatileImage` 的概率就很少，甚至没有，因为数据一直在变。在这种情况下，通过缓存算法禁用 `VolatileImage` 缓存，因为如果缓存的内容一直都得不到使用，也就没有必要将 `BufferedImage` 的内部修改复制到 `VolatileImage` 的内部。另一方面，如果只在自动图像首次创建的时候更新图像数据，那么缓存算法就会重复使用 `VolatileImage` 缓存。

缓存算法是怎样知道什么时候要启用或关闭 `VolatileImage` 缓存呢？以我个人的观点看来，该算法在缓存模式和非缓存模式之间进行切换好像是要取决于像素在一个间隔延迟(该延迟每次都翻一倍)内是否被修改过。注意，由于自动图像的源代码是不可用的，所以在这个问题上，我的理解可能也不对。此外，由于算法还没有文档化，所以在未来的 Java 中，其实现方法可能还要进行改变，以使其在一般情况下也能提高性能。

如果对图像像素数据的修改很少但是修改都具有规律性，那么缓存算法目前的实现可能还会有一些问题。在这种情况下，在大部分时间里，自动图像都是以启用缓存的方式运行的。当像素数据被修改以后，一般都希望缓存算法立即切换到缓存模式，因为马上就会再次修改数据。但是缓存算法却不知道这些，它在从非缓存模式切换到缓存模式之前就要等待一段时间。每次对像素数据进行修改的时候，这种延迟都会增加一倍，因此 `VolatileImage` 硬件加速的优势也就逐渐消失。此外，在非缓存和缓存模式之间的切换对玩家来说，可能是显而易见的，因为在没有外界客观原因影响的情况下，帧速率会突然有一个明显的提高。

如果您不能接受这种缓存算法的行为，或担心将来它会影响您对其实现所做的修改，那么您可以实现自己的自动图像类。如果是这样，我希望您能仔细分析一下我的 `VolatileImageIcon` 和 `ResourceImageIcon` 实现。回想一下前面用来将透明像素的数据从稳定图像转移到默认的自动图像中 `VolatileImage` 所使用的那个“内部技巧”，这个“内部技巧”还没有文档化，也不可

能使用公有的 API 方法。正是由于这个原因，如果您不能真正给出自己的实现技巧的话，可能也就不能在您自己的自动图像实现中使用带透明背景的精灵图像。

然而，一般来说，几乎总是需要修改图像的像素数据，要么是每一帧都修改，要么只在初始化图像时修改一次。在这两种经常发生但是又比较极端的情况中，您可能希望像在默认自动图像中实现的那种缓存算法那样，按照您的想法去实现这些修改：在前面的第一种情况中像 `BufferedImage`，而在前面的第二种情况中又像 `VolatileImage`。很少需要您直接去初始化和操纵 `VolatileImage`。为了得到最佳的执行性能，我推荐您应该使用便利构造函数 `loadAutomaticImage()`，并对这个方法使用透明度参数(`OPAQUE` 或 `BITMASK`)，用它来加载静态的图像，例如精灵和背景瓦片。

5.2 多缓冲和多线程

在重绘组件的过程中，为了防止出现组件表面的不完全更新，一般就使用一种双重缓冲(double-buffering)技术。在双重缓冲中，更新首先被绘制在一个单独的 offscreen 内存缓冲区中，然后当整个组件的更新完成以后，再将这个单独的 offscreen 内存缓冲区中的内容快速地复制到 onscreen 图形缓冲区中。在 AWT 的 `Component` 子类中，`update()`方法就经常会被重写，以便为双重缓冲实现可定制代码，也便于在重新绘制整个背景颜色的时候，消除动画闪烁。但是在 Swing 中，定制的代码往往是多余的，也不需要使用 `update()`方法来解决这种问题，因为 `JComponent` 本身默认的就是双重缓冲。可以关闭双重缓冲，以观察使用类 `JComponent` 的方法 `setDoubleBuffered(boolean)`的效果。

在标准的 Swing 双重缓冲中，线程调度就不再会成为问题，因为只有一个线程：事件分派线程。当组件需要重绘以响应鼠标单击事件或窗口事件的时候，这个线程就能很好进行处理。但是在 Swing 动画中，通常都使用一个单独的线程在一个快速循环中驱动 offscreen 缓冲区进行周期性地更新。如果事件分派线程正在从 offscreen 缓冲区向 onscreen 缓冲区复制数据时，动画显存也同时更新了 offscreen 缓冲区，那么就有可能产生问题。正如第 3 章介绍的一样，`AnimatedComponent` 回避这个问题的方法是：使用 `EventQueue` 类的 `invokeAndWait()`方法将在 `animationThread` 循环中这种调用和事件分派线程的绘制操作进行同步。

但是如果 `animate()`方法要花很长的时间去执行的话，这样处理就还会存在问题。这种问题的发生是有多种原因的，包括在一个非常慢的机器上运行动画、进行复杂的像素计算、要处理大量的数据，或通过一个非常慢的网络连接访问更新以后的数据等。这会使事件分派线程过载，而该线程的过载又会使所有其他的 Swing 事件处理停止下来。例如，重绘处理和关闭窗口的请求可能就会被不定期地延迟一段时间。这意味着用户将不能控制窗口，程序就会表现出一种死锁的状态。

`LoopGovernor` 类 `govern()` 方法的延迟不会产生这种类似的问题，因为它是在 `animationThread` 的内部而不是在事件分派线程中运行的。但是执行精灵更新和绘制操作的 `animate()`方法，还是运行在事件分派线程中，如果运行很慢，就可能会导致一些很严重的问题。

当有非常长的更新和绘制操作时，最好的解决办法可能就是使用三重缓冲(triple-buffering)技术。当有两个非同步的线程一起工作来创建动画的时候，就需要使用三重缓冲技术。第一个线程只是在最快的一个循环中完成活动的 offscreen 缓冲区到 onscreen 缓冲区的复制。第二个

线程负责更新活动的 offscreen 缓冲区。技巧就是第一个线程与第二个线程没有同步，并且可能会在任何时候都试图使用活动的 offscreen 缓冲区。第二个线程通过交替更新一个活动 offscreen 缓冲区，并在完成的时候快速地交换一个指针以将其变为活动的 offscreen 缓冲区的方法，处理这种定时问题。

BufferedAnimatedComponent 是一个 **AnimatedComponent** 子类，该子类使用三重缓冲来防止长时间的更新和绘制阶段妨碍 Swing 事件的及时处理。**BufferedAnimatedComponent** 的用法在 CroftSoft Collection 的 Evolve 程序中演示。

```
package com.croftsoft.core.animation.component;
```

```
[...]
```

```
public class BufferedAnimatedComponent
    extends AnimatedComponent
```

BufferedAnimatedComponent 是 **com.croftsoft.core.animation.component** 包中 **AnimatedComponent** 的一个子类。

```
private boolean      doReset;

private VolatileImage activeImage;

private VolatileImage updateImage;

private Graphics2D    activeGraphics;

private Graphics2D    updateGraphics;

private int           oldCount;

private Rectangle [ ] oldRepaintRegions;

private Rectangle [ ] newRepaintRegions;
```

除了从它的超类 **AnimatedComponent** 中继承来的保护型变量以外，**BufferedAnimatedComponent** 还定义了一些其他的实例化变量。布尔标志 **doReset** 用来表示有两个 offscreen 缓冲区需要改变大小或需要刷新。活动和非活动的 offscreen 缓冲区是分别使用 **activeImage** 和 **updateImage** 进行引用的。正如所期望的一样，图形上下文 **activeGraphics** 属于 **activeImage**，**updateGraphics** 属于 **updateImage**。**oldCount** 数组记录前一帧中重绘请求的数目。**oldRepaintRegions** 数组存储这些重绘请求。**newRepaintRegions** 数组是在当前帧中产生的重绘请求的一个临时存储位置。

```
public BufferedAnimatedComponent (
    ComponentAnimator componentAnimator,
    RepaintCollector   repaintCollector,
    LoopGovernor       loopGovernor )
////////////////////////////////////
{
    super (
```

```

        componentAnimator,
        repaintCollector,
        loopGovernor );

    setDoubleBuffered ( false );

    activeImage = NullVolatileImage.INSTANCE;
}

[...]
```

主构造函数关闭了 Swing 中默认的双重缓冲机制，这是因为当其子类实现了三重缓冲技术时，双重缓冲机制就变成多余的了。由于缓冲区一般都比较宝贵，尤其是当它们扩展到整个屏幕大小的时候，任何人都希望关闭双重缓冲以减少不必要的 offscreen 缓冲区并释放一些内存。然而，这个命令只是禁用 Swing 的 offscreen 缓冲区，缓冲区仍会创建，这是因为缓冲区可能仍然会被除了该类以外的其他 Swing 组件继续使用。实际上，最终可能会有 4 个内存缓冲区：onscreen 显存缓冲区、Swing 的 offscreen 缓冲区、活动 offscreen 缓冲区 activeImage 以及非活动 offscreen 缓冲区 updateImage。

如果您为内存的问题而伤尽脑筋，但是除了 BufferedAnimatedComponent 以外又没有其他 Swing 组件共享 Window，那么您可以使用 javax.swing.RepaintManager 类中的下面命令，安全地移除 Swing 的 offscreen 缓冲区。

```

RepaintManager repaintManager
    = RepaintManager.currentManager ( null );

repaintManager.setDoubleBufferingEnabled ( false );
```

BufferedAnimatedComponent 的主构造函数的最后一个动作是将 activeImage 引用初始化为 com.croftsoft.core.awt.image 包中 NullVolatileImage 类的一个单态实例。NullVolatileImage 是包 java.awt.Image 中 VolatileImage 类的一个空对象子类的实现。在稍后一节的 paintComponent() 方法中会再对其进行介绍。

上面的代码段省略了 4 个便利构造函数，这些构造函数与在前面第 3 章中介绍的超类 AnimatedComponent 中的便利构造函数非常相似。

```

public void init ( )
///////////////////////////////////////////////////
{
    super.init ( );

    addComponentListener (
        new ComponentAdapter ( )
        {
            public void componentResized ( ComponentEvent componentEvent )
            {
                doReset = true;
            }
        }
    );
```



```

    oldRepaintRegions = new Rectangle [ ] { };
    newRepaintRegions = new Rectangle [ ] { };
}

```

该超类通过将doReset标志设为true，就可以对组件大小调整事件进行响应。动画循环周期性地检查这个标志，以决定是否也应该改变offscreen缓冲区的大小。init()方法也将重绘区域存储数组的长度初始化为0。

该init()方法重写超类AnimatedComponent的init()方法。其他的生命周期方法，start()、stop()和destroy()方法，都没有被重写，只是原封不动地直接继承使用。

```

public void paintComponent ( Graphics graphics )
///////////////////////////////////////////////////
{
    graphics.drawImage ( activeImage, 0, 0, null );
}

```

这个方法只是将活动 offscreen 缓冲区复制到 onscreen 显存缓冲区。当事件分派线程内有一个串行化任务时，就会调用这个方法。您应该回想起前一章 AnimatedComponent 超类的实现在 ComponentAnimator 实例中调用了 paint()方法。这种行为被重写了，因为调用 paint()方法将会在事件分派线程中占用较长的时间，并阻塞其他 Swing 事件的处理。可以先作预产生的 activeImage 的一个快速转移。

注意，框架可以在活动 offscreen 缓冲区 activeImage 准备好之前就调用这个方法。之所以会发生这种情况，是由于在产生帧的时候，会在动画循环内部给 activeImage 分派一个有效的对象，而在动画循环启动以前，却可能已经调用了 paintComponent()方法。如果 activeImage 为空，因为代码在使用 activeImage 以前并没有检查它是否为空，所以调用这个方法将会产生一个 NullPointerException 异常。

我们通过使用空对象的方法就可以避免 这个问题。空对象的实现是作为一个占位符使用的，占位符所占的位置就是不想插入代码检查空值，但是又不总能拥有一个有效对象的位置。空对象的子类实现了应该在代码的那个位置上出现的这个对象的类，因此代码就将其视为多态，而不加任何区别地调用它的方法。但是空对象的所有方法都不做任何事情，那些返回值的方法通常也就是返回一个 0 值或空值。

由于在动画过程中，paintComponent()方法是被频繁(通常是一秒几次)调用的，所以不检查空值就显得非常有用，特别是当这个值只是在该方法第一或第二次调用时空值时。正如前面代码给出的一样，BufferedAnimatedComponent 的主构造函数将 activeImage 分配给 NullVolatileImage 的一个单态实例(VolatileImage 的一个空对象实现)。Graphics 类的 drawImage()方法似乎很乐意处理这种长度和宽度均为 0 的 Image 哑元子类，即使它不断地引发 NullPointerException 异常(如果它有空参数的话)。在动画开始的时候，变量 activeImage 后来就会被重新分配给一个有效的 VolatileImage 对象。

您可能会想，为什么我在构造函数中不直接给 activeImage 分配 VolatileImage 的一个小的空实例，而是使用一个特殊的空对象类，例如 NullVolatileImage。这里的问题在于这个 VolatileImage，像它的超类 Image 一样，它只是一个抽象类，因此不能直接进行实例化。它必须使用一个像 java.awt.Component 类中的 createVolatileImage()这样的方法进行创建。但是，在组件刚刚被初始化以后，以及由 Swing 框架第一次显示它以前，或在第一次显示的过程中，立

即调用这个方法，它就会返回空值。如果在游戏实现的过程中，注意到当代码第一次或第二次调用 `paintComponent()` 方法的时候，会引发一个 `NullPointerException` 堆异常，但是当程序运行时间久了以后又没有任何问题了，那么就需要检查这种问题。

```
protected void loop ( )
////////////////////////////////////
{
    doReset = true;
```

`loop()` 方法首先将 `doReset` 标志设为 `true`，以便于标识出将进行初始化的 `offscreen` 缓冲区。

```
while ( animationThread != null )
{
    try
    {
        animateOffscreen ( );

        if ( doReset )
        {
            continue;
        }
    }
```

循环会持续运行，直到 `destroy()` 方法取消对 `animationThread` 的引用。`loop()` 方法首先调用 `animateOffscreen()` 方法，该方法重置 `offscreen` 缓冲区(如果需要的话)，并在非活动的 `offscreen` 缓冲区中执行 `update` 和 `paint` 操作，以产生动画的下一帧。由于这可能需要花费一段时间，所以在循环的时候应该立即检查在这个过程中是否做了一个重置请求。如果有重置请求，它就会重启循环而不显示非法帧。

```
        EventQueue.invokeLaterAndWait ( animationRunner );

        loopGovernor.govern ( );

        if ( stopRequested )
        {
            synchronized ( this )
            {
                while ( stopRequested )
                {
                    wait ( );
                }
            }
        }
    }
    catch ( InterruptedException ex )
    {
    }
    catch ( InvocationTargetException ex )
    {
        ex.getCause ( ).printStackTrace ( );
    }
}
```


上面的这一部分代码与第3章中介绍的超类实现代码相同。

```

if ( activeImage != null )
{
    activeImage.flush ( );

    activeImage = null;
}

if ( updateImage != null )
{
    updateImage.flush ( );

    updateImage = null;
}

if ( updateGraphics != null )
{
    updateGraphics.dispose ( );

    updateGraphics = null;
}

if ( activeGraphics != null )
{
    activeGraphics.dispose ( );

    activeGraphics = null;
}
}

```

循环正常退出的时候，它会释放 offscreen 缓冲区以及它们的图形上下文。完成释放的代码并没有放置在 `destroy()` 方法中，这是因为即使在 `destroy()` 方法完成以后，直到动画循环完成它的最后一次迭代以前，都还有可能要继续使用 offscreen 缓冲区。如果在 `destroy()` 方法完成以前就释放缓冲区，那么就可能会在循环的最后一次迭代中，引发令人讨厌的 `NullPointerException` 异常。

```

protected void animateOffscreen ( )
////////////////////////////////////
{
    if ( doReset )
    {
        if ( activeGraphics != null )
        {
            activeGraphics.dispose ( );
        }

        if ( updateGraphics != null )
        {
            updateGraphics.dispose ( );
        }
    }
}

```

```

    if ( updateImage != null )
    {
        updateImage.flush ( );
    }

```

`animateOffscreen()`方法启动的时候，它首先就检查是否请求了重置。如果已经请求了重置，`animateOffscreen()`方法就释放 `offscreen` 图形上下文和非活动的 `offscreen` 缓冲区。它仍然不会释放活动的 `offscreen` 缓冲区，因为稍后还要在这个方法内使用它。

```

int width = getWidth ( );

int height = getHeight ( );

VolatileImage oldActiveImage = activeImage;

VolatileImage newActiveImage
    = createVolatileImage ( width, height );

if ( newActiveImage == null )
{
    return;
}

```

如果万一由于改变了组件的大小而产生了重置请求，那么就会获取组件的新尺寸，以使 `offscreen` 缓冲区的大小能够调整到与之匹配。原来活动的 `offscreen` 缓冲区就会被保留以便于后面使用，也会重新为它分配实例引用。如果新图像的创建返回空值，那么组件就可能还没有准备好进行显示，该方法就会直接返回而不做任何事情。

```

activeGraphics = newActiveImage.createGraphics ( );

if ( oldActiveImage != null )
{
    activeGraphics.drawImage ( oldActiveImage, 0, 0, null );

    oldActiveImage.flush ( );
}

activeImage = newActiveImage;

```

如果成功创建了新的活动 `offscreen` 缓冲区，我们就要再为它创建一个新的图形上下文，并将原来活动 `offscreen` 缓冲区的像素数据复制给它。我们还要再释放原来的缓冲区。这样做就是为了当正在通过一个可能是相当长的过程产生新像素数据的同时，新缓冲区不会完全为空白。

由于原来的活动 `offscreen` 缓冲区可能会比新的缓冲区还要小，所以组件的某些部分到最后就可能仍然是空白的。以前，通过使用 `Graphics` 类 `drawImage()`方法的变体(该变体允许通过给出宽度和高度参数，将原来缓冲区缩放到新的缓冲区中)，我已经发现了这些空白区域。但是，当组件的大小每次发生改变的时候，这种缩放很容易使屏幕上的信息发生变形，而这种变形往往会使图像在产生下一帧之前，变得不可辨别。正是由于那个原因，当前的实现才没有重新进行调整。

然后，实例变量 `activeImage` 再被分配给工作变量 `newActiveImage`。使用方法变量 `newActiveImage` 而不是直接使用实例变量 `activeImage` 的原因有两点。第一个原因是如果较早对 `createVolatileImage()` 的调用返回空值，使用 `newActiveImage` 能够保证永远不会为 `activeImage` 赋上空值。假定 `activeImage` 有一个空值，即使只是一会儿，也会导致 `paintComponent()` 方法抛出 `NullPointerException` 异常。第二个原因是，我们想等到原来的 `activeImage` 已经复制到新的 `activeImage` 上之后再作转移。这消除了转移的过程中，由于系统窗口事件调用了 `paintComponent()` 方法产生的空白屏幕会短暂闪烁的危险。

```
updateImage = createVolatileImage ( width, height );
```

```
if ( updateImage == null )
{
    return;
}
```

```
updateGraphics = updateImage.createGraphics ( );
```

创建非活动的 `offscreen` 缓冲区，并为它创建一个图形上下文。

```
activeGraphics.setFont ( getFont ( ) );
```

```
updateGraphics.setFont ( getFont ( ) );
```

写到新分配的图形上下文中的默认字体设置为与组件的字体相同。

```
repaintCollector.repaint ( );
```

```
doReset = false;
}
```

由于我们刚刚完成重置请求的处理(该重置请求可能是由于组件大小的改变而产生的)，所以我们要产生一个重绘请求来重绘整个组件。然后再将 `doReset` 标志重置为 `false`，这样在下次循环中，就不需要再次重置 `offscreen` 缓冲区了。

```
if ( stopRequested || animationThread == null )
{
    return;
}
```

```
componentAnimator.update ( this );
```

代码在继续对精灵的位置进行更新之前，它首先检查是否仍然有这样做有必要。如果已经有了一个暂停或退出动画循环的请求，那么这个方法就会立即返回。

```
int count = repaintCollector.getCount ( );
```

```
Rectangle [ ] repaintRegions
    = repaintCollector.getRepaintRegions ( );
```

```
for ( int i = 0; i < count; i++ )
```

```

{
    if ( i == newRepaintRegions.length )
    {
        newRepaintRegions = ( Rectangle [ ] ) ArrayList.append (
            newRepaintRegions, new Rectangle ( repaintRegions [ i ] ) );
    }
    else
    {
        newRepaintRegions [ i ].setBounds ( repaintRegions [ i ] );
    }
}

```

在更新阶段，通过 `RepaintCollector` 收集的重绘请求都被复制到 `newRepaintRegions`。这样做是因为即使 `RepaintCollector` 对象即将被修改，也需要将数据保留起来，以备后用。

```

for ( int i = 0; i < oldCount; i++ )
{
    Rectangle oldRepaintRegion = oldRepaintRegions [ i ];

    repaintCollector.repaint (
        oldRepaintRegion.x,
        oldRepaintRegion.y,
        oldRepaintRegion.width,
        oldRepaintRegion.height );
}

```

在前一个更新阶段收集到的所有重绘请求都被添加到当前更新阶段由 `RepaintCollector` 收集的重绘请求中。这样执行是因为 `offscreen` 缓冲区正在发生改变，它只为每一个其他更新阶段接收重绘。如果没有这样做，精灵就会在它们的身后留下一条糟糕的痕迹。

```

oldCount = count;

Rectangle [ ] tempRepaintRegions = oldRepaintRegions;

oldRepaintRegions = newRepaintRegions;

newRepaintRegions = tempRepaintRegions;

```

将 `oldCount` 变量的值设置为在当前更新阶段中产生的重绘请求的数目，以便于在下一个循环迭代中使用。使用一个临时占位符引用 `tempRepaintRegions`，将 `oldRepaintRegions` 引用和 `newRepaintRegions` 引用进行交换。执行这种交换是因为我们希望将新的重绘请求作为原来的重绘请求保存起来，以便下次使用。我们也希望保存原来的重绘请求，这样就可以重用它们，以存储产生的新重绘请求的数据，而不用再分配一个新的 `Rectangle` 对象数组。

```

count = repaintCollector.getCount ( );

repaintRegions = repaintCollector.getRepaintRegions ( );

```

现在，前一帧里的重绘请求已经与当前更新阶段中的重绘请求在 `RepaintCollector` 中结合在一起，接着就可以获取新请求的数目。`RepaintCollector` 的一个巧妙实现会从前一更新阶段和

当前更新阶段中，合并相互重叠的重绘请求，这样合并以后，重绘请求的数目很可能要比二者之和要小。例如，无论在当前帧还是在前一帧的更新阶段中产生了一个重绘整个组件的重绘请求，在这两个阶段中收集到的所有其他的重绘请求都可能会合并为一个重绘请求。

```
for ( int i = 0; i < count; i++ )
{
    if ( doReset || stopRequested || animationThread == null )
    {
        return;
    }

    updateGraphics.setClip ( repaintRegions [ i ] );

    componentAnimator.paint ( this, updateGraphics );
}
```

绘制更新的区域可能要花较长的一段时间，因此我们在每一个迭代的开始处检查该重绘请求是否依然是必需的。如果是必需的，代码就会只在更新的区域绘制非活动的 offscreen 缓冲区。使用 setClip()可以减少 paint()必须要完成的工作量，这样就可以大大地提高动画的帧速率。

```
if ( updateImage.contentsLost ( ) )
{
    doReset = true;
}
}
```

如果非活动 offscreen 缓冲区的数据丢失，就需要请求缓冲区的重置。loop()方法就会返回，这个标志的状态也会立即通过调用动画循环代码得到检查。

```
protected void animate ( )
///////////////////////////////////////////////////
{
    VolatileImage tempImage = activeImage;

    activeImage = updateImage;

    updateImage = tempImage;

    Graphics2D tempGraphics = activeGraphics;

    activeGraphics = updateGraphics;

    updateGraphics = tempGraphics;
```

animate()方法首先将活动 offscreen 缓冲区的引用与非活动 offscreen 缓冲区的引用交换。缓冲区的相应图形上下文引用也进行同样的交换。引用的这种交换也正是三重缓冲技术的精髓所在。

如果在引用发生交换的时候，paintComponent()方法仍正使用原来的 activeImage，那会发生什么呢？我们知道，这是不可能发生的，因为对 paintComponent()方法的所有调用都是通过事件分派线程串行执行的。由于作这种交换的 animate()方法是使用 EventQueue 类的

invokeAndWait()方法调用的, 所以 paintComponent()方法肯定也是通过事件分派线程串行执行的, 而不可能会重叠。

由于 invokeAndWait()方法在 animationThread 中将动画循环延迟到 animate()方法完成以后, 所以 invokeAndWait()方法也就阻止了 animationThread 跑到事件分派线程的前面。如果这不是问题, 那么动画循环可能就会产生比它们能够显示的速率要快的帧, 并在它被交换的同时, 将其写到 offscreen 缓冲区中。

```
int count = repaintCollector.getCount ( );

Rectangle [ ] repaintRegions
    = repaintCollector.getRepaintRegions ( );

for ( int i = 0; i < count; i++ )
{
    paintImmediately ( repaintRegions [ i ] );
}

repaintCollector.reset ( );
}
```

还会再次使用合并后的重绘请求。以前使用它们绘制非活动的 offscreen 缓冲区。而现在, 它已经交换为活动 offscreen 缓冲区, 于是就使用该请求将数据复制到 onscreen 缓冲区。每一次对 paintImmediately()的调用, 都会导致对 paintComponent()的调用, 而后者负责传输像素数据。

完成的时候就会重新设置 RepaintCollector, 同时它的数据也被全部丢掉。但是别忘了, 在这个迭代过程中收集的重绘请求已经被复制到 oldRepaintRegions 中, 以备在下次迭代中使用。

这一节主要回顾了 BufferedAnimatedComponent 的源代码。在决定是否要使用它或它的超类 AnimatedComponent 时, 请记住使用额外的缓冲区可能要花更多的钱。从前, 我曾遇到过耗尽所有内存, 最终还是导致游戏 applet 崩溃的尴尬问题(当我将组件扩展到全屏的时候)。一般情况下, 您都会希望使用 AnimatedComponent 而不是 BufferedAnimatedComponent, 除非您怀疑 animate()方法中执行更新和绘制操作的时间太多。可能达到这种程度的一个标志就是, 不管怎样优化更新和绘制阶段, GUI 在对用户输入的响应上总是表现得比较迟钝。

5.3 全屏独占模式

以全屏独占模式运行游戏会使游戏充满整个屏幕。除了能够增大动画的区域和增强美学效果以外, 全屏独占模式还需要设置显示模式和防止画面出现锯齿。

5.3.1 启用全屏模式

内核包 java.awt 中的类 GraphicsDevice 里的方法 setFullScreenWindow()的用途就是将游戏置为全屏独占模式。需要提供一个 Window 实例作为这个方法的参数。启用以后, 通过使用空参数调用这个方法, 还可以关闭全屏模式, 返回窗口模式。

有些平台并不支持全屏模式，因此它是通过将游戏窗口放大到整个屏幕的大小来模拟全屏模式的。本章后面介绍的有些游戏技术，例如设置显示模式和防止锯齿，只适合于非模拟的全屏独占模式。可以通过调用 `GraphicsDevice` 的方法 `isFullScreenSupported()` 来确定某一个特定的平台是否支持真的非模拟的模式。

如果游戏是运行在一个无符号 applet 安全沙箱内，请求全屏独占模式可能会从内核包 `java.security` 中抛出 `AccessControlException` 异常。但是屏幕仍然会在这个过程中改变大小。

5.3.2 FullScreenToggler

`com.croftsoft.core.gui` 包中的类 `FullScreenToggler` 可以将一个 `Window` 切换到全屏独占模式，或切换回来。只要用户按下 `Alt+Enter` 组合键或在程序中调用 `toggle()` 方法，就会发生“切换(toggle)”事件。在 `CroftSoft Collection` 中，按下 `Alt+Enter` 组合键可以演示 `FullScreenToggler`。

当提到“全屏模式”而没有使用“独占”这个词的时候，实际上指的是模拟或非模拟的全屏模式。当使用“全屏独占模式”的时候，指的就是真正的非模拟的全屏独占模式，这种模式只在有些平台上可用。类 `FullScreenToggler` 可以用于模拟的和非模拟的这两种全屏模式。

```
package com.croftsoft.core.gui;

import java.awt.*;
import java.awt.event.*;
import java.security.AccessControlException;
import javax.swing.*;

import com.croftsoft.core.lang.NullArgumentException;

public final class FullScreenToggler
    extends AbstractAction
{
    //////////////////////////////////////
    //////////////////////////////////////
    {
```

`FullScreenToggler` 扩展了内核包 `javax.swing` 中的 `AbstractAction`，因此它可以由键盘操作进行触发。

```
public static final String ACTION_KEY_TOGGLE_FULLSCREEN
    = "com.croftsoft.core.gui.FullScreenToggler";
```

键盘的组合键 `Alt+Enter` 将映射到 `String ACTION_KEY_TOGGLE_FULLSCREEN`，而这个 `String` 再映射到 `FullScreenToggler`。

```
private final Window window;
```

类 `JFrame` 是 `Window` 的一个子类，因此这两个类的任何实例都可以通过 `FullScreenToggler` 进行操纵。

```
public static void main ( String [ ] args )
{
    JFrame jFrame = new JFrame ( "Press ALT-ENTER to toggle" );
```

```

jFrame.setDefaultCloseOperation (
    WindowConstants.DO_NOTHING_ON_CLOSE );

jFrame.addWindowListener ( new ShutdownWindowAdapter ( ) );

WindowLib.centerOnScreen ( JFrame, 0.8 );

toggle ( JFrame, true );

monitor ( JFrame );

jFrame.show ( );
}

```

静态 `main()` 方法能够以命令行的方式测试和演示 `FullScreenToggler`。它创建一个大小为屏幕大小百分之八十的窗口，再将该窗体扩大到全屏。用户通过使用 `Alt+Enter` 组合键可以将 `Window` 切换为原始大小(屏幕的百分之八十)。支持类 `ShutdownWindowAdapter` 和 `WindowLib` 与 `FullScreenToggler` 在同一个包中。

```

public static boolean monitor ( JComponent component )
///////////////////////////////////////////////////////////////////
{
    NullPointerException.check ( component );

    Component parent = component;

    while ( parent != null )
    {
        if ( parent instanceof Window )
        {
            KeyStroke keyStroke = KeyStroke.getKeyStroke (
                KeyEvent.VK_ENTER, InputEvent.ALT_MASK, false );

            InputMap inputMap = component.getInputMap (
                JComponent.WHEN_IN_FOCUSED_WINDOW );

            inputMap.put ( keyStroke, ACTION_KEY_TOGGLE_FULLSCREEN );

            inputMap = component.getInputMap ( JComponent.WHEN_FOCUSED );

            inputMap.put ( keyStroke, ACTION_KEY_TOGGLE_FULLSCREEN );

            inputMap = component.getInputMap (
                JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT );

            inputMap.put ( keyStroke, ACTION_KEY_TOGGLE_FULLSCREEN );

            component.getActionMap ( ).put (
                ACTION_KEY_TOGGLE_FULLSCREEN,
                new FullScreenToggler ( ( Window ) parent ) );
        }
    }
}

```



```

        return true;
    }

    parent = parent.getParent ( );
}

return false;
}

```

当 component 参数传递给静态方法 monitor()的时候,这个方法首先就在 Container 的层次结构中向上检索,直到它找到包含这个 component 的 parent Window。对 applet 嵌入在浏览器中的情况而言,可能就不存在 Window。组件的 InputMap 和 ActionMap 被修改过,这样才使 Alt+Enter 组合键的 keyStroke 事件会激活操纵 Window 的 FullScreenToggler。类 InputMap 和 ActionMap 在内核包 javax.swing 中。

```

public static void monitor ( JFrame jFrame )
///////////////////////////////////////////////////////////////////
{
    NullArgumentException.check ( jFrame );

    monitor ( jFrame.getRootPane ( ) );
}

```

前面的 monitor()方法接收 JComponent 类的一个参数。JFrame 本身不是 JComponent 的一个子类,但是它的 JRootPane 是 JComponent 的一个子类。

```

public static void toggle (
    Window window,
    boolean fullScreen )
///////////////////////////////////////////////////////////////////
{
    NullArgumentException.check ( window );

    GraphicsConfiguration graphicsConfiguration
        = window.getGraphicsConfiguration ( );

    GraphicsDevice graphicsDevice
        = graphicsConfiguration.getDevice ( );

    if ( fullScreen )
    {
        try
        {
            graphicsDevice.setFullScreenWindow ( window );

            window.validate ( );

            window.repaint ( );
        }
        catch ( AccessControlException ex )
        {

```

```

    }
}
else
{
    try
    {
        graphicsDevice.setFullScreenWindow ( null );

        window.validate ( );

        window.repaint ( );
    }
    catch ( AccessControlException ex )
    {
    }
}
}

```

如果该方法的参数 `fullScreen` 为 `true`, 那么静态方法 `toggle()` 就会将该 `window` 扩展到全屏。如果 `fullScreen` 为 `false`, 它就会将该 `window` 返回普通模式。

```

public static void toggle ( Window window )
///////////////////////////////////////////////////////////////////
{
    NullPointerException.check ( window );

    GraphicsConfiguration graphicsConfiguration
        = window.getGraphicsConfiguration ( );

    GraphicsDevice graphicsDevice
        = graphicsConfiguration.getDevice ( );

    toggle ( window, graphicsDevice.getFullScreenWindow ( ) != window );
}

```

第二个 `toggle()` 方法与第一个 `toggle()` 方法相似, 只是它没有带有 `fullScreen` 参数。如果屏幕不是全屏模式, 它会将 `window` 切换到全屏模式。如果屏幕是全屏模式, 它就会将屏幕切换回窗口模式。

```

public FullScreenToggler ( Window window )
///////////////////////////////////////////////////////////////////
{
    NullPointerException.check ( this.window = window );
}

///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////

public void toggle ()
///////////////////////////////////////////////////////////////////
{
    toggle ( window );
}

```



```

public void toggle (boolean fullScreen)
///////////////////////////////////////////////////
{
    toggle ( window, fullScreen );
}

```

toggle()实例方法只是简单地委托给那些静态方法。

```

public void actionPerformed ( ActionEvent actionEvent )
///////////////////////////////////////////////////
{
    toggle ( window );
}

```

调用的时候，方法 actionPerformed()会在全屏模式和窗口模式之间进行切换。只要玩家按下了 Alt+Enter 组合键，一般都会调用 actionPerformed()方法。

5.3.3 配置帧

当游戏窗口是全屏模式的时候，您可能希望删除帧的边界和控制条。这也是很合理的，因为屏幕上只有一个游戏窗口，该窗口之后操作系统的桌面是不可见的。删除帧的装饰边框可以提高游戏的外观。当那些没有什么经验的玩家意外地单击窗口控制，中断他们游戏的时候，这样还可以使他们免受这些挫折。

内核包 java.awt 中的类 Frame 为此专门提供了 setUndecorated()方法。这个方法必须在 Frame 第一次显示出来之前进行调用。由于这个限制，当玩家从窗口模式切换到全屏模式时，不能将装饰边框从 Frame 中删除。但是为了达到相同的效果，每次显示模式发生改变的时候，可以创建一个新的 Frame。

类 Frame 还提供了一个 setResizable()方法，当 Frame 是全屏模式的时候，可以使用这个方法将改变窗口大小的功能关闭。您可能希望在游戏玩家将游戏切换到全屏模式时，关闭改变窗口大小的功能，而在玩家将游戏切换回窗口模式的时候，再将这个功能开启。即使不是全屏模式，如果游戏的图形是不能缩放的，您可能也希望关闭这个功能。

闪屏(splash screen)可能就是由于没有可以可缩放图形而造成的。闪屏通常是固定大小的一个高品质的图像，在游戏正在加载和初始化它自身的时候，这个图像会暂时性地越过桌面显示。闪屏 Frame 通常都是未加任何装饰边框的，同时调整大小的功能也是被关闭的。记住，由于闪屏不能被最小化、移动和调整大小，所以对用户来说，在游戏完成加载以前，很难(或可以说是不可能)控制使用他们计算机做其他的事情。

5.3.4 改变显示模式

您可能有位图格式的图形图像，这些图形图像在某一个特定屏幕分辨率和位深度上显示出来的时候效果最好。例如，如果只使用 8 位色的图形，通过将显示模式设为每像素 8 位而不是默认设置的高位色(例如 32 位真彩色)显示，就既可以节省开支，又可以提高帧速率的性能。在这些情况下，可以使用内核包 java.awt 的类 GraphicsDevice 的 setDisplayMode()方法来设置显示器的显示模式。

同一个类中的 `isDisplayChangeSupported()` 在当前的平台上标识设置的显示模式是否可行。注意，一般情况下，在设置显示模式以前，游戏必须是全屏独占模式。

内核包 `java.awt` 的类 `DisplayMode` 是一个数据对象，它存储一个屏幕的高度、宽度、颜色深度和刷新率。类 `GraphicsDevice` 中的方法 `getDisplayModes()` 返回当前平台上的一个可用显示模式的数组。可以在这个数组中循环查找与游戏的图形最兼容的 `DisplayMode`。

5.3.5 DisplayModeLib

内核包 `com.croftsoft.core.gui` 的类 `DisplayModeLib` 是一个操作 `DisplayMode` 对象的静态方法库。它提供了 `matches()` 方法来执行所支持的 `DisplayMode` 与所希望的 `DisplayMode` 的模板匹配。`desiredDisplayMode` 中的 0 值是作为一个通配符使用的。`DisplayModeLib` 也提供了 `print()` 方法来将 `DisplayMode` 对象的值打印到标准的输出上。

5.3.6 GraphicsDeviceLib

内核包 `com.croftsoft.core.gui` 的类 `GraphicsDeviceLib` 是一个操作 `GraphicsDevice` 对象的静态方法库。它提供了 `changeDisplayMode()` 方法，将显示模式改变为当前平台所支持的另外一个显示模式。区分优先次序的理想的 `DisplayMode` 实例数组是以参数的形式给出的。为了减小可能组合的数目，一般需要使用 `DisplayModeLib` 类的模式匹配算法。

```
public static boolean changeDisplayMode (
    GraphicsDevice graphicsDevice,
    DisplayMode [ ] desiredDisplayModes )
////////////////////////////////////
{
    NullPointerException.check ( graphicsDevice );

    NullPointerException.check ( desiredDisplayModes );

    if ( !graphicsDevice.isDisplayChangeSupported ( ) )
    {
        return false;
    }
}
```

如果显示模式发生了改变，该方法就返回 `true`；如果显示模式没有发生改变，该方法就立即返回 `false`。

```
DisplayMode currentDisplayMode = graphicsDevice.getDisplayMode ( );
DisplayMode [ ] supportedDisplayModes
    = graphicsDevice.getDisplayModes ( );
```

获取 `currentDisplayMode` 和 `supportedDisplayModes` 数组。

```
for ( int i = 0; i < desiredDisplayModes.length; i++ )
{
    DisplayMode desiredDisplayMode = desiredDisplayModes [ i ];

    if ( DisplayModeLib.matches (
```



```

        currentDisplayMode, desiredDisplayMode ) )
    {
        return false;
    }

```

如果 `currentDisplayMode` 与其中的一个 `desiredDisplayModes` 匹配, 该方法就直接返回而不会改变任何东西。

```

    for ( int j = 0; j < supportedDisplayModes.length; j++ )
    {
        DisplayMode supportedDisplayMode = supportedDisplayModes [ j ];

        if ( DisplayModeLib.matches (supportedDisplayMode, desiredDisplayMode) )
        {
            graphicsDevice.setDisplayMode ( supportedDisplayMode );

            return true;
        }
    }

    return false;
}

```

如果一个 `supportedDisplayModes` 与一个 `desiredDisplayModes` 相互匹配, 显示模式就会发生改变。

5.3.7 消除锯齿

当动画没有和显示器的刷新率同步的时候, 就会出现一些锯齿, 这是由于正在显示时, 显存正好也被更新所造成的。这锯齿在刚开始时可能不会感觉到。当背景动画移动的时候, 人们就会很明显地感觉到这种锯齿。可以在 CroftSoft Collection 中的示例程序 Sprite 观察这种现象(当瓦片背景移动的时候)。

可以将显示器的刷新率和动画的刷新率同步来消除锯齿。可以使用多缓冲来实现这种同步。当 `offscreen` 缓冲区的绘制完成的时候, 就会更新显示内存的显示指针, 以使 `offscreen` 缓冲区变为 `onscreen` 缓冲区。这种切换正好发生在显示器刷新结束和下次刷新开始之前, 因此更新就得到同步。这种技术被称为页交换技术(page flipping)。这种技术并不是在所有的平台上都可以使用, 通常只在游戏是全屏独占模式时才可以使用。

在有些平台上, 可能需要将像素数据从 `offscreen` 缓冲区复制(或 blitted)到 `onscreen` 缓冲区, 而不是简单地更新显存的显示指针。这最好能够使用硬件加速的显示内存缓冲区来实现, 这样可以提高帧速率。如果这不可用, 可以停止使用没有加速的系统内存。

当动画循环因为绘制 `offscreen` 缓冲区而不能与刷新率保持一致的时候, 可以使用另外一个 `offscreen` 缓冲区。当完成绘制第一个 `offscreen` 缓冲区的时候, 可以立即开始绘制第二个 `offscreen` 缓冲区, 而不用等到第一个 `offscreen` 缓冲区被交换(或 blitted)到 `onscreen` 缓冲区上以后再绘制第二个 `offscreen` 缓冲区。

如果动画运行的速度始终比刷新率要快, 创建第二个 `offscreen` 缓冲区对内存来说就是一种浪费, 因为在显示器的刷新做好准备的时候, 第一个 `offscreen` 缓冲区总是能够完成绘制。注意,

在这种情况下，使用帧速率同步来驱动动画而不使用 `LoopGovernor` 也是可以的。例如，如果将显示模式设置为 85Hz 的刷新率，而动画的刷新率不低于 85Hz，那么精灵每帧一个像素的移动速度正好相当于每秒 85Hz 的刷新率。

可以在类 `Window` 中的 `createBufferStrategy()` 中，以参数的形式，指定缓冲区的数目和期望的缓冲区容量。这个方法的第一种形式只带有一个参数——缓冲区的数目——并尽量使用页交换技术。如果页交换不可用，它就试着使用加速的 `blitting`。如果 `blitting` 也不能使用，它就使用没有加速的 `blitting`。

调用 `createBufferStrategy()` 方法以后，就使用类 `Window` 中的 `getBufferStrategy()` 方法从类 `java.awt.image` 中检索 `BufferStrategy` 的实例。`BufferStrategy` 中的方法 `getDrawGraphics()` 提供 `Graphics` 对象，该对象将 `offscreen` 包装起来。`offscreen` 缓冲区绘制以后，就调用 `BufferStrategy` 中的方法 `show()` 执行交换或 `blit`。

5.3.8 BufferStrategyAnimatedComponent

包 `com.croftsoft.core.animation.component` 中的类 `BufferStrategyAnimatedComponent` 是 `AnimatedComponent` 的一个子类，该类使用了 `BufferStrategy`。在想通过将 `onscreen` 缓冲区更新的频率与显示器的刷新率同步来消除锯齿的时候，就可以使用这个类。

```
package com.croftsoft.core.animation.component;

import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferStrategy;

import com.croftsoft.core.animation.AnimatedComponent;
import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.animation.RepaintCollector;
import com.croftsoft.core.animation.factory.DefaultAnimationFactory;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.util.loop.FixedDelayLoopGovernor;
import com.croftsoft.core.util.loop.LoopGovernor;

public class BufferStrategyAnimatedComponent
    extends AnimatedComponent
    //////////////////////////////////////
    //////////////////////////////////////
{

    protected final BufferStrategy bufferStrategy;

    //

    private final Rectangle componentBounds;

    private final Rectangle clipBounds;
```

与 `AnimatedComponent` 不同，`AnimatedComponent` 依赖于 `paintImmediately()` 方法，而 `BufferStrategyAnimatedComponent` 必须显式地将 `clipBounds` 设置在 `componentBounds` 范围内。

当通过使用 `Integer.MAX_VALUE` 而不是实际组件的范围作出重绘这个组件产生的请求时，就可能会产生一个非常大的 `clipBounds`。

```
public BufferStrategyAnimatedComponent (
    ComponentAnimator componentAnimator,
    RepaintCollector repaintCollector,
    LoopGovernor loopGovernor,
    BufferStrategy bufferStrategy )
////////////////////////////////////
{
    super ( componentAnimator, repaintCollector, loopGovernor );

    NullArgumentException.check (
        this.bufferStrategy = bufferStrategy );

    componentBounds = new Rectangle ( );

    clipBounds = new Rectangle ( );
}
```

必须将 `bufferStrategy` 作为一个参数传递给主构造函数。

```
public BufferStrategyAnimatedComponent (
    ComponentAnimator componentAnimator,
    BufferStrategy bufferStrategy )
////////////////////////////////////
{
    this (
        componentAnimator,
        DefaultAnimationFactory.INSTANCE.createRepaintCollector ( ),
        new FixedDelayLoopGovernor ( 0, 0 ),
        bufferStrategy );
}
```

当固定延迟为 0 的时候，最大的帧速率限制就与显示器的刷新率相等，因为 `BufferStrategyAnimatedComponent` 不可能超过刷新率。理想的情况下，如果动画和图形处理足够快，实际达到的帧速率就会与刷新率相等。

```
protected void animate ( )
////////////////////////////////////
{
    componentAnimator.update ( this );

    int count = repaintCollector.getCount ( );

    Rectangle [ ] repaintRegions
        = repaintCollector.getRepaintRegions ( );

    getBounds ( componentBounds );

    Graphics2D graphics2D
        = ( Graphics2D ) bufferStrategy.getDrawGraphics ( );
```

```

for ( int i = 0; i < count; i++ )
{
    Rectangle repaintRegion = repaintRegions [ i ];

    if ( !componentBounds.intersects ( repaintRegion ) )
    {
        continue;
    }

    Rectangle2D.intersect (
        componentBounds, repaintRegion, clipBounds );

    graphics2D.setClip ( clipBounds );

    componentAnimator.paint ( this, graphics2D );
}

bufferStrategy.show ( );

graphics2D.dispose ( );

repaintCollector.reset ( );
}

```

超类 `AnimatedComponent` 中的 `animate()` 方法被重写为使用 `bufferStrategy`。注意，`componentAnimator` 的 `paint()` 方法并不知道或不关心 `graphics` 参数是从哪里来的。在这个 `animate()` 方法中，它是通过 `bufferStrategy` 使用 `getDrawGraphics()` 方法得来的。当在 `bufferStrategy` 上调用 `show()` 方法的时候，代码就会被挂起，直到在两次刷新之间执行了交换才会恢复。

5.3.9 FullScreenDemo

包 `com.croftsoft.ajgp.grap` 中的类 `FullScreenDemo` 演示了用于更改显示模式和消除锯齿的全屏独占模式的用法。该类以每帧一个像素的速度将墙砖图案滑过屏幕。实际达到的采样帧速率显示在屏幕的左上角。在装有较好显卡的机器上，`FullScreenDemo` 能够成功地与 75Hz 刷新率的显示器达到同步(这时的分辨率为 1280×1024 像素，32 位真彩色)。单击鼠标可以结束这个示例程序。这个程序是使用 Ant 构建目标 `fullscreen` 编译和执行的。

如果全屏独占模式不可用或在当前的配置中不能用，那么 `FullScreenDemo` 就会自动在模拟的全屏模式中运行。当在一个安全沙箱(在这里，通过使用带 `-Djava.security.manager` 命令行参数的 Java 命令将全屏模式禁止)中运行它的时候，就可以观察到底会发生什么事情。

```
java -jar fullscreen.jar 1280 1024 32 75
```

如上面这一行代码所示，我们可以通过依次将宽度、高度、位深度和刷新率以参数的形式传给命令行的方法，选用一个理想的显示模式。0 值只表示一个通配符，表示显示模式可以设置为任何支持的值。

```

package com.croftsoft.ajgp.grap;

import java.awt.*;
import java.awt.image.BufferStrategy;

```



```

import java.awt.event.*;
import java.io.*;
import java.security.*;
import javax.swing.*;

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.animation.AnimatedComponent;
import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.animation.ComponentPainter;
import com.croftsoft.core.animation.animator.FrameRateAnimator;
import com.croftsoft.core.animation.animator.NullComponentAnimator;
import com.croftsoft.core.animation.animator.TileAnimator;
import com.croftsoft.core.animation.component
    .BufferStrategyAnimatedComponent;
import com.croftsoft.core.animation.painter.ColorPainter;
import com.croftsoft.core.animation.painter.NullComponentPainter;
import com.croftsoft.core.awt.image.ImageLib;
import com.croftsoft.core.gui.BufferCapabilitiesLib;
import com.croftsoft.core.gui.DisplayModeLib;
import com.croftsoft.core.gui.FullScreenToggler;
import com.croftsoft.core.gui.GraphicsDeviceLib;
import com.croftsoft.core.gui.WindowLib;
import com.croftsoft.core.lang.lifecycle.Lifecycle;

public final class FullScreenDemo
    extends JApplet
    implements ComponentAnimator, Lifecycle
{
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    {

    [...]

    private static final DisplayMode [ ] DESIRED_DISPLAY_MODES
    = new DisplayMode [ ] {
        new DisplayMode ( 640, 480, 8, 85 ),
        new DisplayMode ( 640, 480, 8, 75 ),
        new DisplayMode ( 640, 480, 8, 70 ),
        new DisplayMode ( 640, 480, 8, 0 ),
        new DisplayMode ( 640, 480, 0, 0 ),
        new DisplayMode ( 800, 600, 8, 85 ),
        new DisplayMode ( 800, 600, 8, 75 ),
        new DisplayMode ( 800, 600, 8, 70 ),
        new DisplayMode ( 800, 600, 8, 0 ),
        new DisplayMode ( 800, 600, 0, 0 ) };

```

如果显示模式的值没有以命令行参数的形式给出，FullScreenDemo 就会试着使用默认的 DESIRED_DISPLAYED_MODES。我喜欢首先试用较高的刷新率，因为我发现 60Hz 的刷新率是很让人烦躁的。如果不支持 640×480 的分辨率，可以使用 800×600 的分辨率，但是显示将会变慢一些，因为每帧中显示的像素要多一些。

```
private static final int BUFFER_COUNT = 2;
```

在假定动画能够和显示器的刷新率保持同步的前提下，就只会使用两个缓冲区：onscreen 缓冲区和 offscreen 缓冲区。

```
private static final String MEDIA_DIR = "media/fullscreen/";
```

```
private static final String TILE_IMAGE_FILENAME
    = "clear_brick_32x32.png";
```

8 位色的瓦片图像有着完全透明的背景。没有存储任何中间半透明的值，所以硬件加速会发挥作用。

```
private final BufferStrategy bufferStrategy;
```

直到在 init() 方法中需要 BufferStrategy 的时候为止，BufferStrategy 一直是以实例变量的形式存储的。

```
private AnimatedComponent animatedComponent;
```

```
private ComponentPainter brickColorPainter;
```

```
private ComponentAnimator brickTileAnimator;
```

```
private ComponentAnimator frameRateAnimator;
```

根据全屏独占模式在运行时是否可用，animatedComponent 类要么是 AnimatedComponent，要么是它的子类 BufferStrategyAnimatedComponent。brickColorPainter 将背景绘制为红色。brickTileAnimator 驱动一个墙砖图案滑过背景。frameRateAnimator 采样并显示实际达到的帧速率。

```
public static void main ( String [ ] args )
///////////////////////////////////////////////////
{
    System.out.println ( INFO );

    JFrame jFrame = new JFrame ( );

    jFrame.setUndecorated ( true );

    GraphicsConfiguration graphicsConfiguration
        = jFrame.getGraphicsConfiguration ( );

    GraphicsDevice graphicsDevice
        = graphicsConfiguration.getDevice ( );
```

方法 main() 首先创建一个未加装饰的 JFrame。然后通过它的 graphicsConfiguration 检索它的 graphicsDevice。

```
System.out.println ( "Initial display mode:" );

DisplayModeLib.print ( graphicsDevice.getDisplayMode ( ) );
```


最初显示模式的值(在它们改变以前)会打印到标准的输出上。

```
BufferStrategy bufferStrategy = null;
```

`bufferStrategy` 初始化为 `null` 值。如果全屏独占模式不可用或不允许使用,那么 `bufferStrategy` 就会保留它自己的 `null` 值,同时就会使用模拟的全屏模式。

```
try
{
    graphicsDevice.setFullScreenWindow ( JFrame );
```

如果运行在一个无符号 `applet` 安全沙箱的限制以内,这个方法就会抛出 `AccessControlException` 异常。奇怪的是,在它抛出 `SecurityException` 子类实例以前,它会成功地切换到模拟的全屏模式上。

```
JFrame.validate ( );
```

```
JFrame.repaint ( );
```

我不能完全肯定在将 `JFrame` 设置为全屏以后,就真的需要验证和重绘 `JFrame`。如果您遇到这样的问题,在调用 `setFullScreenWindow()`以后,计算机出现了灰屏,而解决这种灰屏的方法只有一种,就是重新启动您的计算机,那么您可以试一试。

```
System.out.println ( "\nisDisplayChangeSupported: "
    + graphicsDevice.isDisplayChangeSupported ( ) );

DisplayMode [ ] desiredDisplayModes = DESIRED_DISPLAY_MODES;

if ( args.length == 4 )
{
    desiredDisplayModes = new DisplayMode [ ] {
        new DisplayMode (
            Integer.parseInt ( args [ 0 ] ),
            Integer.parseInt ( args [ 1 ] ),
            Integer.parseInt ( args [ 2 ] ),
            Integer.parseInt ( args [ 3 ] ) ) );
    }
}
```

想得到的 `DisplayMode` 值可以解析为几个命令行参数。

```
boolean displayModeChanged
    = GraphicsDeviceLib.changeDisplayMode (
        graphicsDevice, desiredDisplayModes );

if ( displayModeChanged )
{
    System.out.println ( "\nNew display mode:" );

    DisplayModeLib.print ( graphicsDevice.getDisplayMode ( ) );
}
```

如果改变了显示模式,那么新的显示模式的值就会打印到标准的输出上。

```

if ( graphicsDevice.isFullScreenSupported ( ) )
{
    System.out.println ( "\nFull-screen exclusive mode supported" );

    JFrame.setIgnoreRepaint ( true );

    JFrame.createBufferStrategy ( BUFFER_COUNT );

    bufferStrategy = JFrame.getBufferStrategy ( );

    BufferCapabilitiesLib.print (
        bufferStrategy.getCapabilities ( ) );
}

```

如果不支持全屏独占模式，就没有必要处理由窗口事件产生的重绘命令。创建一个 `bufferStrategy`，并使用 `com.croftsoft.core.gui` 包中的 `BufferCapabilitiesLib` 类的一个静态方法，将它的功能打印到标准的输出上。

```

else
{
    System.out.println (
        "\nFull-screen exclusive mode unsupported" );
}
}
catch ( AccessControlException ex )
{
    System.out.println ( "\nFull-screen exclusive mode not allowed" );
}

FullScreenDemo fullScreenDemo
    = new FullScreenDemo ( bufferStrategy );

```

如果全屏独占模式不可用，以构造函数的参数传递来的 `bufferStrategy` 可能就会为 `null`。

```

JFrame.setContentPane ( fullScreenDemo );

JFrame.show ( );

fullScreenDemo.init ( );

fullScreenDemo.start ( );
}

```

在调用 `init()` 方法以前，就将 `JFrame` 处理成可以显示的，这样在 `init()` 方法内也就能够加载兼容图像。接着就初始化并启动动画。

```

public FullScreenDemo ( BufferStrategy bufferStrategy )
///////////////////////////////////////////////////
{
    this.bufferStrategy = bufferStrategy;

    brickColorPainter = NullComponentPainter .INSTANCE;
}

```



```

brickTileAnimator = NullComponentAnimator.INSTANCE;

frameRateAnimator = NullComponentAnimator.INSTANCE;
}

```

这些空对象的单独实例，只在下面这种情况——正好在构造函数以后，在调用 `init()` 方法以前，有一个对 `paint()` 方法的调用——才用作临时的占位符。这可以防止抛出 `NullPointerException` 异常。

```

[...]

public void init ( )
///////////////////////////////////////////////////
{
    if ( bufferStrategy == null )
    {
        animatedComponent = new AnimatedComponent ( this );
    }
    else
    {
        animatedComponent
            = new BufferStrategyAnimatedComponent ( this, bufferStrategy );
    }
}

```

使用 `bufferStrategy` 在全屏独占模式中运行的这种决定，是在运行时根据机器的性能、操作系统以及系统上正在运行的应用程序等因素做出的。

```

animatedComponent.addMouseListener (
    new MouseAdapter ( )
    {
        public void mousePressed ( MouseEvent mouseEvent )
        {
            FullScreenToggler.toggle (
                WindowLib.getParentWindow ( animatedComponent ), false );

            System.exit ( 0 );
        }
    }
);

```

鼠标单击将会关闭全屏独占模式，而且不会进行进一步的清理就立即退出程序。

```

Container contentPane = getContentPane ( );

contentPane.setLayout ( new BorderLayout ( ) );

contentPane.add ( animatedComponent, BorderLayout.CENTER );

validate ( );

```

`animatedComponent` 被添加到这个 `JApplet` 子类的 `contentPane` 以后，就调用 `validate()` 刷新布局。

```

animatedComponent.init ( );

brickColorPainter = new ColorPainter ( Color.RED );
try
{
    Image brickTileImage = ImageLib.loadAutomaticImage (
        MEDIA_DIR + TILE_IMAGE_FILENAME,
        Transparency.BITMASK,
        animatedComponent,
        getClass ( ).getClassLoader ( ),
        null );

    Icon brickTileIcon = new ImageIcon ( brickTileImage );

    brickTileAnimator = new TileAnimator (
        0, 0, brickTileIcon, ( Shape ) null, 1, 1 );
}
catch ( IOException ex )
{
    ex.printStackTrace ( );
}

frameRateAnimator = new FrameRateAnimator ( this, Color.ORANGE );
}

```

ComponentPainter 和 **ComponentAnimator** 实例被初始化，替换了那些空对象单态实例。

```

public void start ( ) { animatedComponent.start ( ); }

public void stop ( ) { animatedComponent.stop ( ); }

public void destroy ( ) { animatedComponent.destroy ( ); }

```

这个示例中一直都没有使用 `stop()` 和 `destroy()` 方法，因为鼠标单击事件导致了程序非正常终止。

```

public void update ( JComponent component )
///////////////////////////////////////////////////
{
    brickTileAnimator.update ( component );

    frameRateAnimator.update ( component );
}

public void paint (
    JComponent component,
    Graphics2D graphics2D )
///////////////////////////////////////////////////
{
    brickColorPainter.paint ( component, graphics2D );

    brickTileAnimator.paint ( component, graphics2D );
}

```



```
frameRateAnimator.paint ( component, graphics2D );  
}
```

update()方法和paint()方法都是很简单。

5.3.10 使用独占模式的顾虑

我对使用全屏独占模式始终有些顾虑。一方面，它允许改变显示模式，它也能够消除锯齿。但是，另一方面，它不能在无符号 applet 安全沙箱内部正常发挥作用，也不是在所有的平台上都可以使用全屏独占模式。记住，使用全屏独占模式不需要提高帧速率的性能。如果没有遇到画面有锯齿的情况，并且也不需要改变显示模式的话，就完全不需要使用全屏独占模式。相反，可以更简单地使用与屏幕大小相同而不带任何装饰边框的帧。如果选择使用全屏独占模式，那就确保程序在模拟的全屏模式中也能够很好地运行，就像在 FullScreenDemo 中介绍的一样。

5.4 小结

对那些带有透明背景的硬件加速图像来说，在将图像加载到内存中的时候，请使用 ImageLib 类的静态方法 loadAutomaticImage()，而不是直接实例化 VolatileImage。如果 update() 方法或 paint() 方法太慢，以至于妨碍了在事件分派线程中及时地处理用户输入，那么请使用 BufferedAnimatedComponent 来代替 AnimatedComponent。如果遇到了锯齿或需要改变显示模式，请使用全屏独占模式，但是同时要保证游戏在没有全屏独占模式时也能够很好地运行。

5.5 参考文献

Feldman, Ari. "Designing for Different Display Modes." Chapter 2 in *Designing Arcade Computer Game Graphics*. Plano, TX: Wordware Publishing, 2000.

Kesselman, Jeffrey. "Understanding the AWT Image Types." 2001.

Martak, Michael. "Full-Screen Exclusive Mode API." <http://java.sun.com/docs/books/tutorial/extra/fullscreen/>.

Sun Microsystems. "High Performance Graphics." http://java.sun.com/products/java-media/2D/perf_graphics.html.

Sun Microsystems. "The VolatileImage API User Guide." <ftp://ftp.java.sun.com/docs/j2se1.4/VolatileImage.pdf>.

第 6 章 持久数据

早睡早起，能使人健康、富有和聪明。

—— 本杰明·富兰克林

简单地说，持久性就是将数据保存到硬盘的能力。在游戏编程中，在持久数据方面，我的观点是，您应该将您在这方面的 workload 降到最小。您不应该在这个方面花费太多的精力去做优化和调整。游戏玩家也不会太在意数据的格式或磁盘的存储能力。但是，他们更关心的是可视化方面，例如精灵的行为以及用户界面。

很多编程的工作都要将对象编为定制的二进制数据格式，然后再将对象转换回来。在编写新程序的时候，不是这样不断地进行这种重复性的工作，而是使用数据对象类的实例，这些实例可以直接以对象的形式进行保存，再以对象的形式重新加载到程序中。这一章准备介绍一些很好使用的静态库方法，这些方法会为您完成这种任务。还要讨论游戏应用程序经常使用的一些持久性机制，它们会检查游戏玩家数据文件的变化，并缓存下载下来的文件。

6.1 数据格式

这一节将介绍 4 个持久数据格式，这些格式都很容易使用：对象串行化、属性文件、XML 和图像格式。因为编码器、解码器或解析器在 Java 内核库中都可用，使得这每一种格式都优于定制的数据格式。本节还会介绍使用随机种子产生静态数据。

6.1.1 对象串行化

将一个活动数据对象转换为扁平化字节数组的一个最快方法就是 Java 编程语言所专用的标准对象串行化机制。

```
public static byte [ ] compress ( Serializable serializable )
    throws IOException
////////////////////////////////////
{
    NullPointerException.check ( serializable );

    ByteArrayOutputStream byteArrayOutputStream
        = new ByteArrayOutputStream ( );

    save ( serializable, byteArrayOutputStream );

    return byteArrayOutputStream.toByteArray ( );
}
```


上面给出的 `com.croftsoft.core.io` 包中 `SerializableLib` 类的便利方法 `compress()`，将实现了 `java.io.Serializable` 接口的所有对象转换为压缩的字节数组。记住，接口 `Serializable` 只是一个语义接口。语义接口并没有定义方法或常量；它只是表示某一些意思，或称为语义上的，它应该添加到实现类——这样它就可被串行化。

```
public static void save (
    Serializable serializable,
    OutputStream outputStream )
    throws IOException
    //////////////////////////////////////
{
    NullPointerException.check ( serializable );

    NullPointerException.check ( outputStream );

    ObjectOutputStream objectOutputStream = null;
    try
    {
        objectOutputStream = new ObjectOutputStream (
            new GZIPOutputStream (
                new BufferedOutputStream ( outputStream ) ) );

        objectOutputStream.writeObject ( serializable );
    }
    finally
    {
        if ( objectOutputStream != null )
        {
            objectOutputStream.close ( );
        }
        else
        {
            outputStream.close ( );
        }
    }
}
```

`compress()`方法依赖同一个类中的可重用静态方法 `save()`，这个方法将串行化和压缩的对象数据写入 `OutputStream`。注意，`OutputStream` 包装在 `BufferedOutputStream` 中，`BufferedOutputStream` 包装在 `GZIPOutputStream`，而 `GZIPOutputStream` 又包装在 `ObjectOutputStream` 中。这每一个包装类都会在将它接收到的数据传递给包装它的类之前，修改这些数据。

内核包 `java.io` 中的 `ObjectOutputStream` 类的方法 `writeObject()` 将内存里活动的 `Serializable` 对象转换为扁平化的字节流。内核包 `java.util.zip` 中 `GZIPOutputStream` 类会压缩这些字节流以减少存储空间。内核包 `java.io` 中的 `BufferedOutputStream` 类将写操作缓冲到压缩和串行化的数据的目标 `OutputStream`，这样就可以一次写多个字节块，而不用每次只写一个字节。这样在写文件的时候，就可以明显地提高性能，因为一次写多个字节块降低了对底层操作系统写操作调用的次数。

```

public static Serializable load ( InputStream inputStream )
    throws ClassNotFoundException, IOException
    //////////////////////////////////////
{
    NullPointerException.check ( inputStream );

    ObjectInputStream objectInputStream = null;

    try
    {
        objectInputStream
            = new ObjectInputStream (
                new GZIPInputStream (
                    new BufferedInputStream ( inputStream ) ) );

        return ( Serializable ) objectInputStream.readObject ( );
    }
    finally
    {
        if ( objectInputStream != null )
        {
            objectInputStream.close ( );
        }
        else
        {
            inputStream.close ( );
        }
    }
}

```

相应的 load()方法的执行过程正好相反, 它从 `InputStream` 中返回一个经过解压缩和反序列化以后的对象。注意, 在假定返回的对象实现了这个 `Serializable` 接口的前提下, 返回的对象在返回之前, 会首先会类型转换为 `Serializable` 接口。

```

public class GameData
    implements java.io.Serializable
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 1L;

    //

    public int health;

    public int wealth;

    public int wisdom;

    //////////////////////////////////////
    //////////////////////////////////////
}

```


我有时喜欢采用一些简单的形式，就像上面的示例代码一样。对象通常被定义为封装的数据和操作这些数据的方法。由于类 `GameData` 没有声明任何方法，实例变量也都是公共变量，所以从一般意义来讲，不能将它看作对象。一旦将这个数据对象加载到内存，我通常就会将这个数据转换为一个真正的对象，这个真正的对象将会用适当的 `accessor` 方法和 `mutator` 方法来封装并保护这个数据。

注意，除了实现 `Serializable` 接口以外，类 `GameData` 还声明了常量 `serialVersionUID`，其中 `UID` 是惟一标识符(unique identifier)的缩写。如果使用的是具有惟一性的包和类名称的组合，那么 `serialVersionUID` 这个值可以任意选取，因此我通常最初只选用数字0。当想强制使用一个与以前保存数据的版本不相容的数字时，才改变这个值，通常采用的方式是逐个递增1。如果试用一个不同的 `serialVersionUID`，将一个原来的数据加载到修改后类的一个实例，`ObjectInputStream` 就会注意到这种变化，并引发一个异常。如果该类发生了改变，但是 `serialVersionUID` 并没有改变，`ObjectInputStream` 就会尽力让原来的数据能够适合该类的新版本，删除一些不再使用的数据，并将以前没有使用的新变量设置为默认的空值或0值。

下面的两行示例代码说明在 Java 中使用对象串行化技术保存和加载对象数据其实是非常简单的事情。

```
SerializableLib.save ( gameData, outputStream );

[...]
```

```
gameData = ( GameData ) SerializableLib.load ( inputStream );
```

注意，从 `load()` 方法中返回的对象(在使用它操作数据以前)通常都会转换成一个更具体的类或接口。

6.1.2 属性

虽然对象串行化功能很强大，但是有时您也希望能够以人类可读的文本格式来加载和保存数据。这样做的一个很好的理由就是允许游戏的开发人员和设计者改变游戏的内容。例如，武器的杀伤范围可能就是以普通文本配置文件的形式存储的。如果游戏的设计者不需要程序员的帮助就可以通过修改资源文件中的参数来调整游戏，那么游戏的开发就可以进行得更快。一些聪明的游戏玩家非常喜欢能够轻易对 boss(老板)的图像更改文件名称的能力，也许他们会将图像更改为他真正上司的图像。

```
#Game Data
#Tue Mar 25 15:18:22 CST 2003
wisdom=18
wealth=99
health=10
```

最初，您可能会考虑使用 `java.util.Properties` 类的 `store()` 和 `load()` 方法以普通文本格式存储名值对。但是，我并不推荐这种方法，因为数据字段通常总是以类 `String` 实例的形式返回的。虽然在简单字段类型(例如整数或 `String` 类型的名称)中解析值是很简单的事情，但是处理复杂对象的值可能就比较麻烦。在试图以属性值的方式存储一个有层次结构的对象图表时，更是如此。

6.1.3 XML

以人类可读的格式，使用名值对的属性文件存储数据的另一种方法就是 XML。下面就是 XML 示例文件的内容。

```
<gameData>
  <health>10</health>
  <wealth>99</wealth>
  <wisdom>18</wisdom>
</gameData>
```

不是使用 `java.util.StringTokenizer` 编写自己定制的对象解析器，从属性文件中解析出有层次结构的数据，而是采用 XML 代替文本格式，并使用与 Java 的核心 API 一起提供的一个标准 XML 解析器来做这种解析。除了有现有解析器支持以外，XML 数据格式还有两个优于其他文本格式的优点。第一个优点是它是自解释的。每一个层次上数据元素的值都是用标记包装起来的，这个标记标明该数据是什么。上面示例 XML 代码就说明了将数据打上标记的方法，就连非程序员也可以很容易地理解这些数据，使用一个简单的文本编辑器就可以对这些数据进行修改。

第二个优点是，可以编写一个模式，该模式会定义和验证数据。当游戏的设计者和画面设计师使用强制数据模式的 XML 编辑器修改 XML 数据的时候，可以减少运行时 bug 的数目。还有一些产品，例如 NETBRYX Technologies 的 XML Quik Builder，可以读取 XML 模式定义 (XML schema definition, XSD) 文件，并能够以 GUI 的形式提供给数据项¹。例如，如果模式将一个元素限制在只有 3 个可选的值，GUI 可能就会给出一个具有 3 个选项的下拉列表。在我看来，这比为游戏设计者和画面设计师创建您自己定义的 GUI 内容编辑器要简单一些。但是，作为程序员来说，这确实需要一些专门的技术和 XML Schema 标准。

可以通过内核包 `javax.xml.parsers` 中的类使用 Simple API for XML (SAX) 或 Document Object Model (DOM) 解析器，但是我更喜欢使用内核包 `java.beans` 中的 `XMLEncoder` 和 `XMLDecoder`。原因是 `XMLEncoder` 和 `XMLDecoder` 可以直接保存和加载对象，就像 `ObjectOutputStream` 和 `ObjectInputStream` 处理 `Serializable` 对象一样。SAX 和 DOM 解析器需要一个中间数据表示形式，这通常就意味着必须编写定制的代码，将数据绑定到对象的表示形式上。Java Architecture for XML Binding (JAXB) 的 API 通过自动产生执行这种数据-对象绑定的代码²解决了这个问题。但遗憾的是，JAXB 并不能够通用，至少在核心 J2SE API 中不能使用。

```
XMLDecoder xmlDecoder
= new XMLDecoder ( new BufferedInputStream ( inputStream ) );

GameConfig gameConfig = ( GameConfig ) xmlDecoder.readObject ( );
```

我们将这里的这个 `XmlDecoder` 示例代码与上面的 `ObjectInputStream` (前一节中介绍的) 进行比较。注意，像对象串行化一样，必须转换返回的对象；与对象串行化不一样，数据不需要进行解压缩。数据压缩会将人类可读的文本数据变为二进制数据。

1 <http://www.editxml.com/QuikBuilder.aspx>

2 <http://java.sun.com/xml/jaxb/>


```

public class GameData
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

private int health;

private int wealth;

private int wisdom;

///////////////////////////////////////////////////
// accessor methods
///////////////////////////////////////////////////

public int getHealth ( ) { return health; }

public int getWealth ( ) { return wealth; }

public int getWisdom ( ) { return wisdom; }

///////////////////////////////////////////////////
// mutator methods
///////////////////////////////////////////////////

public void setHealth ( int health ) { this.health = health; }

public void setWealth ( int wealth ) { this.wealth = wealth; }

public void setWisdom ( int wisdom ) { this.wisdom = wisdom; }

```

使用 XMLEncoder 和 XMLDecoder 时，数据对象类不需要实现 Serializable 接口，但是它确实需要遵循 Java Beans 标准。从本质上说，这意味着需要为所有的数据定义 accessor 方法和 mutator 方法，并需要提供一个没有参数的构造函数，由构造函数设置默认的数据值。在上面的示例中，不带任何参数的构造函数是隐式定义的，因为没有定义其他的构造函数，整数数据值被初始化为默认的值 0。

```

package com.croftsoft.ajgp.data;

import java.beans.XMLDecoder;
import java.beans.XMLEncoder;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.Serializable;

import com.croftsoft.core.lang.Testable;

public final class SerializableGameData
    implements Serializable, GameData, Testable
///////////////////////////////////////////////////
///////////////////////////////////////////////////

```

```

{

private static final long serialVersionUID = 1L;

//

private int health;

private int wealth;

private int wisdom;

////////////////////////////////////
// static methods
////////////////////////////////////

[...]

////////////////////////////////////
// no-argument constructor method
////////////////////////////////////

public SerializableGameData ( )
////////////////////////////////////
{
    setHealth ( DEFAULT_HEALTH );

    setWealth ( DEFAULT_WEALTH );

    setWisdom ( DEFAULT_WISDOM );
}

////////////////////////////////////
// accessor methods
////////////////////////////////////

public int getHealth ( ) { return health; }

public int getWealth ( ) { return wealth; }

public int getWisdom ( ) { return wisdom; }

////////////////////////////////////
// mutator methods
////////////////////////////////////

public void setHealth ( int health )
////////////////////////////////////
{
    if ( health < MINIMUM_HEALTH )
    {
        throw new IllegalArgumentException ( "health < minimum" );
    }
}

```



```

    }

    this.health = health;
}
public void setWealth ( int wealth )
////////////////////////////////////
{
    if ( wealth < MINIMUM_WEALTH )
    {
        throw new IllegalArgumentException ( "wealth < minimum" );
    }

    this.wealth = wealth;
}

public void setWisdom ( int wisdom )
////////////////////////////////////
{
    if ( wisdom < MINIMUM_WISDOM )
    {
        throw new IllegalArgumentException ( "wisdom < minimum" );
    }

    this.wisdom = wisdom;
}

```

如果创建了一个遵循 Java Beans 标准的数据对象类，可能也会将它变成 `Serializable` 的。可以使用 `ObjectOutputStream` 或 `XMLEncoder` 将上面示例类 `SerializableGameData` 扁平化为一个文件。

```

package com.croftsoft.ajgp.data;

public interface GameData
////////////////////////////////////
////////////////////////////////////
{

    public static final int DEFAULT_HEALTH = 10;

    public static final int DEFAULT_WEALTH = 99;

    public static final int DEFAULT_WISDOM = 18;

    //

    public static final int MINIMUM_HEALTH = -10;

    public static final int MINIMUM_WEALTH = 0;

    public static final int MINIMUM_WISDOM = 3;

    //////////////////////////////////////
}

```

```
// accessor methods
////////////////////////////////////

public int getHealth ( );

public int getWealth ( );

public int getWisdom ( );

////////////////////////////////////
// mutator methods
////////////////////////////////////

public void setHealth ( int health );

public void setWealth ( int wealth );

public void setWisdom ( int wisdom );
```

在这个示例中，`SerializableGameData` 实现了一个称为 `GameData` 的接口，该接口定义了 `accessor` 方法和 `mutator` 方法。当使用 `readObject()` 方法从二进制字节流中(无论是从 `ObjectInputStream` 还是从 `XMLDecoder` 中)将对象加载进来以后再进行类型转换时，我都会将它转换成 `GameData` 接口，而不是转换成具体的类 `SerializableGameData`。如果将游戏应用程序编写为使用接口引用访问数据，那么不需要改变应用程序的代码就可以改变数据对象的实现。如果决定升级数据对象的实现时，这种实现可能非常有用。例如，`GameData` 接口的另一种实现可能会被称为 `RandomAccessFileGameData` 或 `RelationalDatabaseGameData`，这些实现使用了一种不同的持久性机制。

在我刚开始编写新游戏时，常常首先使用一些像 `SerializableGameData` 这样简单的接口，将所有数据都加载到内存，并使用简单的对象串行化或 Java Beans XML 编码解码机制将它们保存到磁盘。如果由于某些原因，后面需要改变我的持久性机制时，我就可以很容易地改变持久性机制，因为所有代码的接口引用的使用，事实上已经将实际使用的持久性机制抽象出来。

```
public static final int DEFAULT_HEALTH = 10;

[...]

public static final int MINIMUM_HEALTH = -10;
```

注意，接口也为最小值和默认值定义了一些常量。在接口内部而不是在类的内部定义常量通常都是一种很好的方法。这样，实现该接口的类就可以自动继承这些值，而不必去扩展一个抽象或具体的类，也不必将某一个对象实例化。我们应该记得，Java 支持多接口继承，但是不支持多抽象类或多具体类的继承。

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
  <object class="com.croftsoft.ajgp.data.SerializableGameData">
    <void property="health">
      <int>-1</int>
```



```

    </void>
  </object>
</java>

```

在 XMLEncoder 向外部的数据流写入某个对象时，它会不厌其烦地记录下各个属性的值，这些属性值与由不带参数的构造函数设置的默认值相同。当 XMLEncoder 读入对象时，它首先会使用不带参数的构造函数将这个对象实例化。在 XML 数据中显式给出的任何值都会使用 mutator 方法进行设置。所有其他值都保持其默认值。这能够充分地减小需要存储在 XML 数据文件中数据的数量，尤其是有很多带有值的数据对象，这些数据对象通常又都使用其默认值的时候。例如，当持久性游戏世界内部的大多数妖怪在遭到玩家的伤害以后，都会通过“治疗”恢复其健康值。在上面的示例 XML 数据中，只有属性值 health 不是在没有参数的构造函数中设置的默认值，因此这个属性就被记录下来。

```

public static void main ( String [ ] args )
///////////////////////////////////////////////////
{
    System.out.println ( test ( args ) );
}

public static boolean test ( String [ ] args )
///////////////////////////////////////////////////
{
    try
    {
        final int TEST_HEALTH = -1;

        GameData gameData1 = new SerializableGameData ( );

        System.out.println ( "health = " + gameData1.getHealth ( ) );
        System.out.println ( "wealth = " + gameData1.getWealth ( ) );
        System.out.println ( "wisdom = " + gameData1.getWisdom ( ) );

        gameData1.setHealth ( TEST_HEALTH );

        ByteArrayOutputStream byteArrayOutputStream
            = new ByteArrayOutputStream ( );

        XMLEncoder xmlEncoder = new XMLEncoder ( byteArrayOutputStream );
        xmlEncoder.writeObject ( gameData1 );

        xmlEncoder.close ( );

        byte [ ] xmlBytes = byteArrayOutputStream.toByteArray ( );

        System.out.println ( new String ( xmlBytes ) );

        XMLDecoder xmlDecoder
            = new XMLDecoder ( new ByteArrayInputStream ( xmlBytes ) );
    }
}

```

```

GameData gameData2 = ( GameData ) xmlDecoder.readObject ( );

System.out.println ( "health = " + gameData2.getHealth ( ) );

System.out.println ( "wealth = " + gameData2.getWealth ( ) );

System.out.println ( "wisdom = " + gameData2.getWisdom ( ) );

return gameData2.getHealth ( ) == TEST_HEALTH
    && gameData2.getWealth ( ) == DEFAULT_WEALTH
    && gameData2.getWisdom ( ) == DEFAULT_WISDOM;
}
catch ( Exception ex )
{
    ex.printStackTrace ( );
    return false;
}
}

```

上面给出的静态测试方法成功地演示了游戏数据的 Java Beans XML 编码和解码过程。它也用来产生前面给出的示例 XML 的输出。

```

import com.croftsoft.core.lang.Testable;

public final class SerializableGameData
    implements Serializable, GameData, Testable

```

当像上面给出的方法这样，在类中包含一个静态测试方法的时候，也就意味着该类是为每晚自动化代码库测试准备的，这种测试是通过使用包 `com.croftsoft.core.lang` 中的语义标记接口 `Testable` 标记这个类的方法实现的。有关这种单元测试技术的更多信息，请参考 `Testable` 的 javadoc。

```

public SerializableGameData ( )
///////////////////////////////////////////////////
{
    setHealth ( DEFAULT_HEALTH );

    setWealth ( DEFAULT_WEALTH );

    setWisdom ( DEFAULT_WISDOM );
}

```

不带参数的构造函数将参数传递给 `mutator` 方法。它采用这种方法，而不是由它自己直接设置这些私有变量，能够让它不会绕开 `mutator` 方法里定义的数据范围检查。在这个示例中，如果值越界，例如低于预定义的最小值，`mutator` 方法会抛出 `IllegalArgumentException` 异常。这有助于发现错误，尤其是偶尔将一个默认的常量更改为一个非法值的时候。当合法数据的范围发生改变，而加载的原数据在新合法数据范围以外的時候，这个不带参数的构造函数也会引发 `Exception`。数据实例变量也被声明为 `private`，这样就不会绕过 `mutator` 方法中的范围检查。

```

public SerializableGameData (
    int health,
    int wealth,

```



```

    int wisdom )
    ////////////////////////////////////////////
    {
        setHealth ( health );

        setWealth ( health );

        setWisdom ( wisdom );
    }

    public SerializableGameData ( )
    ////////////////////////////////////////////
    {
        this (
            DEFAULT_HEALTH,
            DEFAULT_WEALTH,
            DEFAULT_WISDOM );
    }

```

您可能会考虑再添加一个主构造函数，该主构造函数带有参数，就像上面的示例一样。如果这样，请确保也同时包含一个不带参数的构造函数。如果没有这样做，Java Beans XML 的编码和解码就不能工作。在这种情况下，不带参数的构造函数可能会委托给主构造函数来将需要复制代码的数量降到最小。

```

GameData gameData = new SerializableGameData ( );

gameData.setHealth ( 4 );

gameData.setWealth ( 200 );

gameData.setWisdom ( 16 );

```

但是，在改变数据的时候，创建一个带参数的构造函数可能带来一些问题。假设决定改变参数的数目或它们的值，比如说将 `health`、`wealth` 和 `wisdom` 改变为 `goldPieces`、`armorClass` 和 `hitPoints`。如果新方法的参数具有相同或相似的数据类型——在这个示例中是基本数据类型(整数)，并且参数的数目相同——在这个示例中是 3 个参数，那为原来构造函数编写的应用程序代码在编译的过程中就会产生错误。但是，如果仅仅提供了一个没有参数的构造函数，使用您代码库的那些开发人员就会被强制使用上面示例代码中给出的 `mutator` 方法来设置这些初始值。使用 `mutator` 方法设置值通常与先后顺序无关。例如，它首先不关心是否调用了 `setHealth()` 或 `setWealth()`。

防止与不匹配的方法参数相关的一些错误的更多信息，请参考下面的“命名表示法”。

命名表示法

```

GameData gameData = new SerializableGameData (
    wealth => 2,
    health => 4 );

```

我对 Java 编程语言的一个愿望就是希望它能拥有我在 Ada 编程语言中发现的一个特性，在 Ada 编程语言中，当传递方法的参数值时，可以可选地定义参数的名称。该表示法使用箭头(=>)，这种箭头不能与大于等于符号相混淆(>=)。这被称为“命名表示法”，以与现在在 Java 中只能使用的“位置表示法”相区别。没有命名的参数(例如假设代码示例中的 wisdom)都被假定为带有一个默认的值。命名表示法使代码对方法签名的更改具有一定的“抵抗力”，例如重新对参数排序，以及删除或添加参数，尤其是相同类型的那些参数时。所有的这些都能使运行时的错误大大减少，生产力得到很大提高，可靠性也得到增强。

```
GameData gameData = new SerializableGameData (
    10,    // health
    99,    // wealth
    18 ); // wisdom
```

命名表示法也有助于将代码自动归档。在上面的示例，我使用了人类可读的标志标识方法的参数，这样在 Java 编程语言中编写代码的时候，就可以获取相同的效果。当参数的顺序改变以后，调试代码应该更加容易——如果您的目的就是这样的话。如果命名表示法在 Java 中得到支持，这种检查就会在编译的过程中自动执行。

几年以前，我曾使用 Swing 较早期的 beta 版编写过代码，其中有一个这样的方法：该方法有列和行(就按列和行的顺序)的整数值作为它的参数。人们习惯说“宽和高”，而不说“高和宽”，而更通常的说法是“行和列”而不是“列和行”。后来 API 发生了变化，改为以行和列这样的顺序给出，我的代码仍然可以很好地编译和运行，但是显示的轴线却颠倒了过来——长和宽颠倒了过来。如果可以在 Java 编程语言中使用命名表示法的话，就可以防止这种问题的发生。

6.1.4 瓦片地图图像

可以使用图形图像文件来存储瓦片地图数据。在这种情况下，游戏设计者使用图像编辑器来绘制要被研究区域的地图。然后程序员就将该地图图像加载到内存中，并将像素转换为地图数据。例如，给定坐标(x, y)位置上的一个绿色像素可以表示地图上的一片草地，而褐色的像素可以表示山脉。

通常您会考虑使用的两个标准的图像格式是图形交换格式(Graphics Interchange Format, GIF)和可移植网络图形(Portable Network Graphics, PNG)格式。这两种格式都是使用无损压缩的方式进行压缩的，这种压缩方式使它们成为一种很理想的图像格式。在这两种图像之间，我更喜欢使用 PNG。PNG 的压缩好像比 GIF 更好，它似乎还具有很多其他图像格式的优点。

PNG 是一个可扩展的文件格式，这个格式可以用于光栅图像的无损的、可移植的和良好的压缩存储。PNG 可以免费替换 GIF，并可以提供 TIFF 的很多通用方法。它支持索引色、灰阶色和真彩色的图像，并为透明处理提供了可选的 Alpha 通道。采样深度的范围从每分量 1 位到每分量 16 位(对 RGB 能够达到 48 位，对 RGBA 能够达到 64 位)。

——World Wide Web 联盟³

3 <http://www.w3.org/Graphics/PNG/>

使用瓦片地图图像的第1个优点是,它们避免了为了进行游戏内容的创作而创建定制的地图或瓦片编辑器的这种需求。地图可能是画在纸上的,我们需要使用彩色扫描仪将它们扫描到计算机。常用的图像编辑程序,例如 Windows MS Paint 或功能更强大的 GIMP,都可以用来创作瓦片地图。

第2个优点可能就是图像编辑器比定制的游戏关、地图以及瓦片编辑器的功能更加强大,特征也更加丰富。您需要从一个正方形背景瓦片上构造出一个大而完美的圆形水池吗?那就画一个圆环并将其填充上蓝色。想用树的形状清楚地表达某一个神秘的消息吗?那就用一种不常用的字体绘制一些文本,并将这些文本染上绿色。想剪切、粘贴、旋转、倾斜、填充、拖放和放大吗?那就在您的图像编辑器中使用这些已经成熟的特性吧。

第3个优点就是数据是可视的。该数据可以使用任意图像浏览器,以图像的方式简单地进行查看,以得到瓦片地图的全景或侧景。假定某个平台只可以将表示瓦片的褐色区域显示为黑色的像素,但是对布局而言,能够显示整个全景图就足够了。

图 6-1 给出了一个示例瓦片地图的图像,该图像长和宽都是 100 个像素。图 6-2 给出了这个示例瓦片地图图像的长和宽都是 40 个像素。正如由一个任意颜色到瓦片映射的函数所决定的一样,只要在瓦片地图图像中有一个青色的像素,就会使用这个特殊的瓦片。图 6-3 给出的是最终的游戏世界的一部分,该图像长和宽都是 4000 个像素(40×100)。

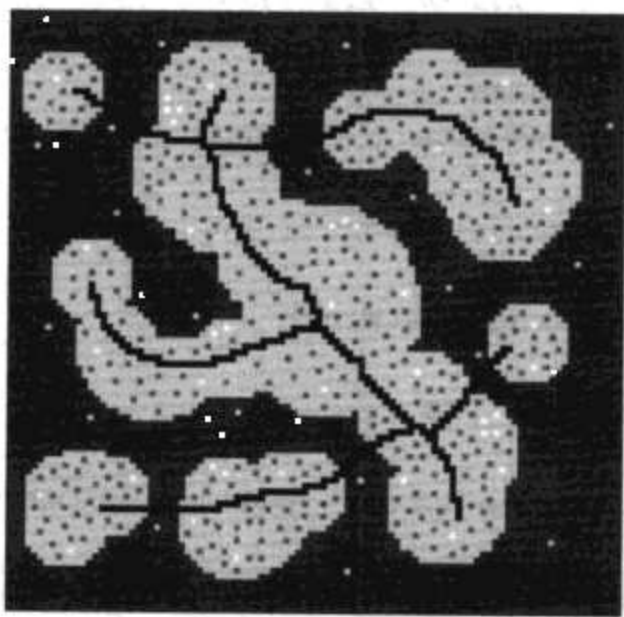


图 6-1 100×100 瓦片地图图像



图 6-2 40×40 瓦片图像

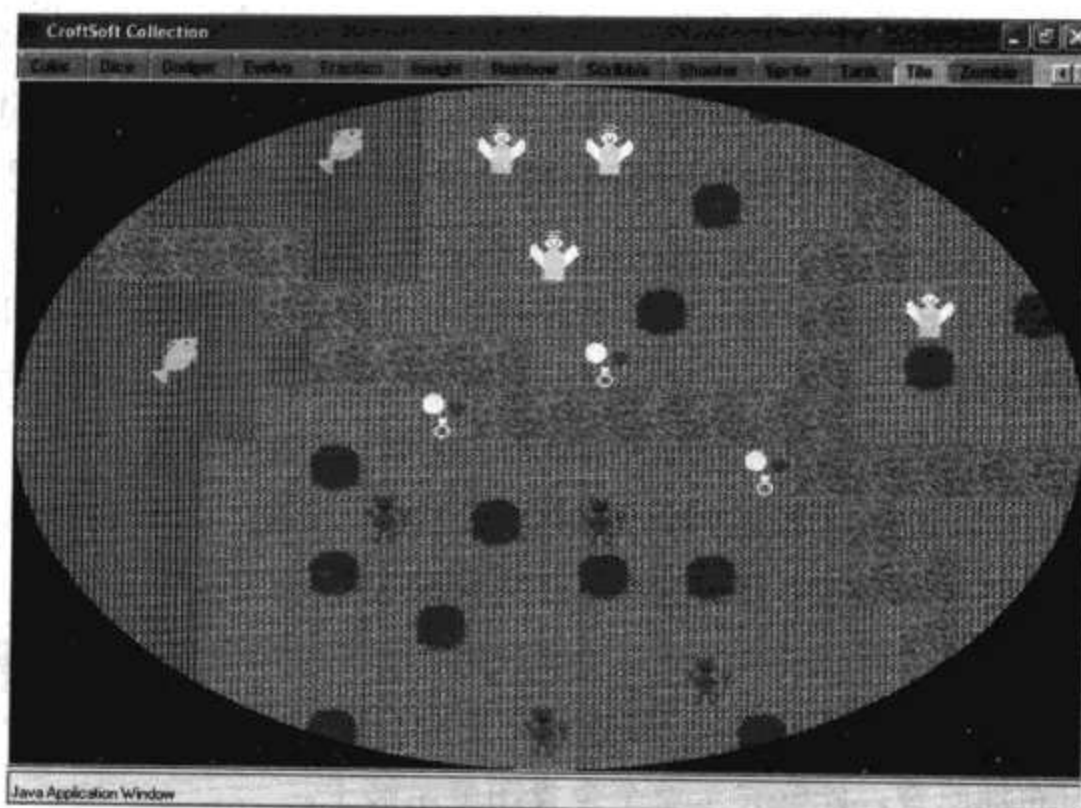


图 6-3 4000×4000 瓦片世界

```

public static TileData loadTileDataFromImage (
    String      filename,
    ClassLoader classLoader )
    throws IOException
    //////////////////////////////////////
{
    BufferedImage bufferedImage
        = ImageLib.loadBufferedImage ( filename, classLoader );

    int rows = bufferedImage.getHeight ( );

    int columns = bufferedImage.getWidth ( );

    int [ ] palette = new int [ 0 ];

    byte [ ] [ ] tileMap = new byte [ rows ] [ columns ];

    for ( int row = 0; row < rows; row++ )
    {
        for ( int column = 0; column < columns; column++ )
        {
            int rgb = bufferedImage.getRGB ( column, row );

            boolean inPalette = false;

            for ( int i = 0; i < palette.length; i++ )
            {
                if ( palette [ i ] == rgb )
                {
                    tileMap [ row ] [ column ] = ( byte ) i;

                    inPalette = true;
                }
            }
        }
    }
}

```



```

        break;
    }
}

if ( !inPalette )
{
    tileMap [ row ] [ column ] = ( byte ) palette.length;

    palette = ( int [ ] ) ArrayList.append ( palette, rgb );
}
}

return new TileData ( palette, tileMap );
}

```

可以在 `com.croftsoft.apps.tile` 包中的类 `TileData` 中找到方法 `loadTileDataFromImage`。该方法将以 `BufferedImage` 的形式加载瓦片地图图像。然后它使用 `BufferedImage.getRGB()` 以 `ARGB(alpha-red-green-blue)` 整型值的形式读入颜色分量。在瓦片地图图像中发现的每一个颜色都会被追加到 `palette` 数组中——如果数组中没有这个颜色的话。`tileMap` 矩阵的值指向调色板内索引的位置。由于 `tileMap` 值是字节类型，所以该调色板的范围限制在 256 种不同的颜色以内。

注意，如果从瓦片图像(该图像是通过将画在纸上的地图扫描到计算机而创建的)中加载瓦片数据，那么在使用之前，可能需要使用图像编辑器清除瓦片地图图像的一个位。例如，如果绿色像素表示树的位置，那么在扫描以后，可能会在原始的瓦片地图中得到很多渐变色(绿色)。图像编辑器或定制的程序可能会习惯将这些不同的渐变色归为所需要的某一个特定的绿色。

6.1.5 随机种子

考虑一下只使用 8 个字节就可以有效地存储无限多个游戏数据的这种情况。这其中的技巧就是使用随机数生成器。别忘了，所有的随机数产生算法实际上都是伪随机的。它们确定性的使用一个算法，从某一个给定点编制成一个数字的无限序列。这个算法被设计成产生通过任何统计测试都表现为随机分布的数字。从某一些以前的点上重新启动随机数生成器，所产生的随机序列就会重复。

它们开始是基于种子值的。如果使用 `java.util.Random` 类默认的不带参数的构造函数，它就会基于当前的时间和许多其他参数，产生一个惟一的种子。但是，也可以将一个常量种子值作为随机数生成器的构造函数参数，来保证随机数生成器每次都产生相同序列的伪随机数字。

随着选用种子的值和在概率映射函数中赋给想要发生事件的概率或不想发生事件概率的不同，就可以得到一个数量无限的游戏数据，游戏每次运行的时候，这些数据都相同，就像它们是存储在某个巨大的资源文件中一样。但是，在这种情况下，只需要以 8 个字节的长度存储随机数生成器的种子和与该种子相关的数据产生代码。使用这 8 个字节，就能够存储 2^{64} 个不同且都具有无限长度的序列中的一个序列。

当然，这种方法也有不好的一面，那就是在能够产生既具有可运行性又具有娱乐性的游戏关卡或地图之前，可能需要仔细检查很多不同的种子，或要很好地调整概率映射函数。假设使用随机数生成器产生迷宫数据。自动生成的有些迷宫可能就没有办法保证迷宫是连通的。在这个示例中，可以手动检查随机产生的迷宫或编写一个算法来自动进行这种检查，然后再将满足可运行标准的那些种子保存起来。

```
public static byte [ ] [ ] generateRandomTileMap (
    Random    random,
    int [ ] palette,
    int      rows,
    int      columns,
    int      smoothingLoops )
////////////////////////////////////
{
    byte [ ] [ ] tileMap = new byte [ rows ] [ columns ];

    for ( int row = 0; row < rows; row++ )
    {
        for ( int column = 0; column < columns; column++ )
        {
            int paletteIndex = random.nextInt ( palette.length );

            tileMap [ row ] [ column ] = ( byte ) paletteIndex;
        }
    }

    for ( int i = 0; i < smoothingLoops; i++ )
    {
        for ( int row = 0; row < rows; row++ )
        {
            for ( int column = 0; column < columns; column++ )
            {
                int left = column > 0 ? column - 1 : columns - 1;

                int right = column < columns - 1 ? column + 1 : 0;

                int up = row > 0 ? row - 1 : rows - 1;

                int down = row < rows - 1 ? row + 1 : 0;

                byte [ ] neighbors = {
                    tileMap [ row ] [ column ],    // center
                    tileMap [ row ] [ left ],      // west
                    tileMap [ row ] [ right ],     // east
                    tileMap [ up ] [ column ],     // north
                    tileMap [ up ] [ left ],       // north west
                    tileMap [ up ] [ right ],      // north east
                    tileMap [ down ] [ column ],   // south
                    tileMap [ down ] [ left ],     // south west
                    tileMap [ down ] [ right ] };  // south east
            }
        }
    }
}
```



```
        tileMap [ row ] [ column ]  
            = neighbors [ random.nextInt ( neighbors.length ) ];  
    }  
}  
  
return tileMap;  
}
```

TileData 类中的 generateRandomTileMap() 方法从调色板中随机地为瓦片分配值。然后它再使用一个平滑算法创建色标“岛屿”。这个颜色聚集算法的过程是这样的：在每一个瓦片位置上进行重复迭代，并随机地将它交换到它最相邻的一个颜色上(交换的概率是 8/9)。更大的 smoothingLoops 值就会导致更大的聚集。如果有太多的平滑操作，最终的游戏世界必然就只有一个颜色。

图 6-4 给出了一部分随机产生的地形图像。使用了具有两种颜色的调色板：蓝色代表海洋，绿色代表草地。随机种子值是 0，平滑循环的数目是 1000。在较慢的机器上，循环 1000 次需要花一定的时间。与使用最大压缩来存储和加载数据的使用类似，我在这里已经决定牺牲初始的处理时间以将数据存储降到最小。

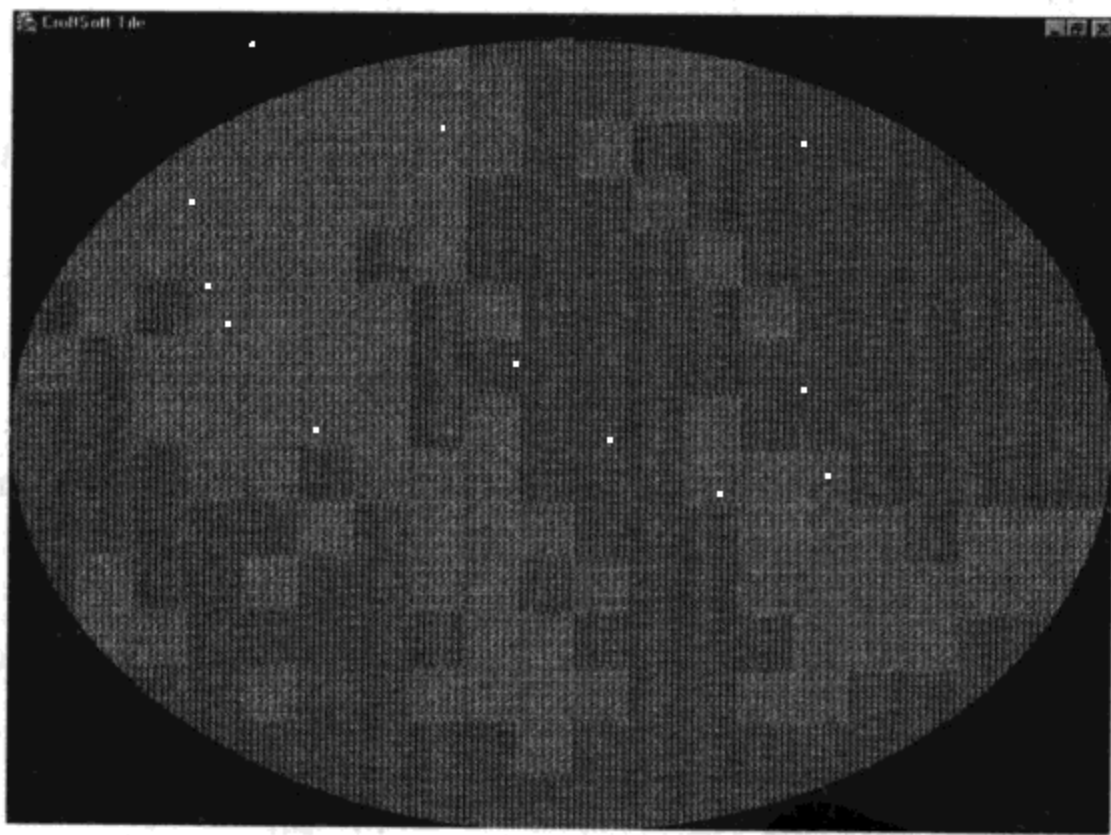


图 6-4 随机产生的地形图像

6.2 持久性机制

这一节将介绍维护数据持久性的各种机制和它们在游戏编程上的适用性。这些机制通常都与数据的格式无关，因为它们存储压缩的二进制文件就像存储文本一样简单。

6.2.1 JAR 资源文件

对静态的游戏内容而言，最好的持久性机制可能就是将数据和代码一起存储在 JAR 文件中。这包括资源文件，例如图形、音频和音乐循环的多媒体文件；使用的所有文本文件，例如积分、故事和非玩家人物(nonplayer character)对话；以及原始的游戏地图或游戏关的数据。

记住，JAR 实际上就是一个压缩的 zip 文件，只是换了另外一个名称而已，因此在将它们包含在归档文件之前压缩这些文件是没有任何问题的。只是这样做的话将会使调试变得更加困难，也不能很好地减小 JAR 文件的大小。即使是多余的，我通常还是会预压缩串行化的对象二进制数据格式的文件，因为我经常首先使用 `SerializableLib.save()` 方法(该方法默认时是使用 GZIP 进行压缩的)创建这些资源文件。

```
*.au -k 'b'
[...]
*.dat -k 'b'
[...]
*.zip -k 'b'
```

对于串行化的对象二进制数据格式文件，我选用了 .dat 文件扩展名。为了保证这种类型的资源文件，在通过以自动纯文本行尾置换的方式存储在 CVS 仓库中时不会造成混淆，我就在 CVSROOT/cvswrappers 文件中包含了 .dat 文件扩展名，将其标识为二进制文件。上面给出的就是这样的一个示例。在这个示例中，我还为众所周知的标准二进制格式，例如音频文件的 AU 和压缩文件的 zip，包含了通用的文件扩展名。有关这一技术的更详细的信息，请参考附录 B。

在 JAR 文件中存储静态数据方面，我最喜欢的一个特性就是对读数据方面没有限制，因为它是和代码打在一个包中的。无论游戏是部署为无符号的 applet、Java Web Start 应用程序还是桌面应用程序，都可以访问该数据而不会抛出安全访问方面的异常。

```
InputStream inputStream
    = getClass ( ).getClassLoader ( ).getResourceAsStream ( DATA_FILENAME );
```

内核包 `java.lang` 中类 `ClassLoader` 的方法 `getResourceAsStream(filename)` 会返回一个 `InputStream`，可以使用 `InputStream` 来访问存储在 JAR 文件中的资源数据。幸运的是，不需要担心 JAR 文件在硬盘驱动器上的位置，只需要知道相对于归档根目录的文件路径。如果是在与编译代码相同的一个 JAR 文件内部保存资源文件，就可以先使用 `Object.getClass()`，再使用 `Class.getClassLoader()` 来获得适当的 `ClassLoader` 实例，以达到此目的。

```
public static Serializable load (
    ClassLoader classLoader,
    String filename )
    throws ClassNotFoundException, IOException
    //////////////////////////////////////
{
    NullPointerException.check ( classLoader );

    NullPointerException.check ( filename );

    InputStream inputStream
        = classLoader.getResourceAsStream ( filename );
```



```

    if ( inputStream == null )
    {
        return null;
    }

    return load ( inputStream );
}

```

如果找不到资源的话，ClassLoader 的方法 `getResourceAsStream()` 就会返回空值。如果发生了这种情况，`load()` 方法也会跟着返回空值。如果找到了资源，它就会通过将检索到的 `InputStream` 传递给上面代码中给出的这个串行化的对象 `load()` 的方式，将资源分配给这个 `load()` 方法。基本上，这个方法所做的所有事情就是将一个特定的持久性机制分配给一个文件格式，在这里，就是从一个静态 JAR 文件(在这个文件中，数据是 GZIP 格式的二进制数据)中加载一个串行化的对象。

可以注意到，在 `SerializableLib` 类中并没有相应的 `save(classLoader,filename)` 方法。尽管 JAR 文件在加载静态游戏内容方面功能很强大，但是它仍然只是一种只读的操作，为了维持静态数据，例如最新的玩家人物(player character, PC)数据或新的最高分记录，需要使用下面几种辅助技术中的某一种技术。

6.2.2 用户主目录文件

对不受安全限制(限制对硬盘驱动器的访问)约束的游戏而言，直接将数据写到文件或直接从文件读取数据可能是最好的持久性机制。可以使用内核包 `java.io` 中的类 `FileInputStream` 和类 `FileOutputStream` 来实现这种操作。

```

public static Serializable load ( String filename )
    throws ClassNotFoundException, IOException
    //////////////////////////////////////
{
    NullPointerException.check ( filename );

    return load ( new FileInputStream ( filename ) );
}

```

`SerializableLib` 这个便利方法将一个 `FileInputStream` 实例传递给前面讨论过的 `load(InputStream)` 方法。只要指定这个文件名称，对象就可以被加载到内存中。

```

public static void save (
    Serializable serializable,
    String filename,
    boolean makeDirs )
    throws IOException
    //////////////////////////////////////
{
    NullPointerException.check ( serializable );

    NullPointerException.check ( filename );

    if ( makeDirs )

```

```

    {
        FileLib.makeParents ( filename );
    }

    save ( serializable, new FileOutputStream ( filename ) );
}

```

相应的 `save()` 方法提供了一个选项来创建文件名称的目录路径——如果需要的话。我发现这一点很好用，因为我经常希望能够将数据文件放在还不存在的一个偏僻目录中。我喜欢使用一个全球惟一性的路径，例如 `.croftsoft/tank/`，其中第一个目录是公司的名称，而第二个目录使用了游戏的名称。公司名称前面的一个句点表示在使用基于 Unix 的操作系统，例如 Linux、Solaris 和 Mac OS X 的时候，该目录应该是隐藏的。

所有的目录和文件名称都使用小写字母，而 Java 源代码文件除外。在开始这样做以前，我经常要花费很多时间来跟踪那些难以捉摸的 bug，这些 bug 最终会成为与区分大小写相关的问题。这种问题经常会在将代码从不区分大小写的部署环境(例如 Windows)，转移到区分大小写的部署环境(例如 Web 或基于 Unix 的服务器)的时候发生。

```
/home/croft/.croftsoft/dodger/dodger.dat
```

```
C:\Documents and Settings\David\.croftsoft\dodger\dodger.dat
```

我将用户特定的游戏数据放在用户主目录内，因为这是得到备份的目录。该路径取决于操作系统、操作系统的版本和它的配置。上面给出了两个示例，第一个是适用于 Red Hat Linux 操作系统的，而第二个是适用于 Windows XP 操作系统的。

```
String userHomePath = System.getProperty ( "user.home" );
```

使用 `System.getProperty()` 方法获取用户主目录的路径并将它作为您路径的另一部分是非常简单的一件事情。如果试图在无符号的 applet 或 Java Web Start 应用程序的安全沙箱限制以内使用这个方法，它就会引发一个异常。别忘了，在这些开发环境中，您没有对用户硬盘驱动器进行读和写的访问权限，因此，在这些环境中不能获取用户主目录的路径实际上也没有什么意义的。

```

public static void save (
    Serializable serializable,
    String         latestFilename,
    String         backupFilename )
    throws IOException
////////////////////////////////////
{
    NullPointerException.check ( serializable );

    NullPointerException.check ( latestFilename );

    NullPointerException.check ( backupFilename );

    File latestFile = new File ( latestFilename );

    if ( latestFile.exists ( ) )

```



```

{
    File backupFile = new File ( backupFilename );

    if ( backupFile.exists ( ) )
    {
        backupFile.delete ( );
    }
    else
    {
        FileLib.makeParents ( backupFilename );
    }

    latestFile.renameTo ( backupFile );
}

save ( serializable, latestFilename );
}

```

我有时担心在保存游戏数据时，这种功能会失败。被中断的保存操作不但会破坏当前游戏的持久性，而且还会破坏以前的游戏数据——如果以前的游戏数据和当前的游戏数据是保存在同一个文件名称下。为了防止发生这种事情，上面列出的 `save()` 方法就带了两个文件名称参数——`latestFilename` 和 `backupFilename`。在 `latestFilename` 中保存当前游戏数据以前，`save()` 方法将会将以前保存的游戏数据更名为 `backupFilename`。如果在保存的过程中，这种功能失败了，那么原来的数据还是受到了保护。

```

public static Serializable load (
    String primaryFilename,
    String fallbackFilename )
    throws ClassNotFoundException, IOException
    //////////////////////////////////////
{
    NullPointerException.check ( primaryFilename );

    NullPointerException.check ( fallbackFilename );

    try
    {
        return load ( primaryFilename );
    }
    catch ( FileNotFoundException ex )
    {
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }

    return load ( fallbackFilename );
}

```

相应的 load() 方法将会首先试图加载以 primaryFilename 的形式保存的数据。如果由于某些原因这种加载失败了，它就会试着加载以前以 fallbackFilename 形式保存的游戏数据。在这个方法中，我将 latestFilename 更名为 primaryFilename，将 backupFilename 更名为 fallbackFilename，因为也可以用不同的方式来使用这个方法。除了加载保存的游戏数据或它的备份数据以外，还可以使用这个方法加载用户的配置文件，或者默认的配置文件(如果用户没有修改过配置文件)。

6.2.3 JNLP 持久性

如果游戏被部署为无符号的 Java Web Start 应用程序，可以使用 JNLP 持久性服务来保存用户数据。像 Web 浏览器中的 cookie 一样，JNLP 客户端(例如 Java Web Start)提供了一种在不侵犯用户安全机制的前提下，可以存储一定大小的持久性数据的机制。这些松饼(muffin)可能还相当大，因为 JNLP 规范要求客户端的实现至少要为每个应用程序提供 128K 的存储空间。对用户特定的数据，例如最高分和保存的游戏数据的这种有限的需求而言，我发现这种小量的存储空间相当可贵。

与 muffin 存储区域关联的 URL 必须与代码库的 URL 在同一个层次上，或在路径层次中高出代码库的 URL。例如，如果从 <http://croftsoft.com/arcade/> 中下载代码，就可以使用 <http://croftsoft.com/arcade/data> 或 http://croftsoft.com/user_preferences 这样的 URL 来存储和检索数据，而不能使用 <http://croftsoft.com/arcade/data/> 或 <http://croftsoft.com/arcade/tank/> 这样的 URL。记住，尽管使用了 muffin 的 URL(该 URL 看起来好像是服务器的地址)，但是数据毕竟是存储在客户端的。

```
public URL createFileContentsURL ( String fileContentsSpec )
    throws MalformedURLException, UnsupportedOperationException
    //////////////////////////////////////
{
    return new URL ( getCodeBase ( ), fileContentsSpec );
}

public URL getCodeBase ( )
    throws UnsupportedOperationException
    //////////////////////////////////////
{
    try
    {
        BasicService basicService = ( BasicService )
            ServiceManager.lookup ( "javax.jnlp.BasicService" );

        return basicService.getCodeBase ( );
    }
    catch ( UnavailableServiceException ex )
    {
        throw ( UnsupportedOperationException )
            new UnsupportedOperationException ( ).initCause ( ex );
    }
}
```


虽然 URL 总是相对于代码库的，但是 `com.croftsoft.core.jnlp` 包中 `JnlpServicesImpl` 类的便利方法 `createFileContentsURL()` 还是将代码库 URL 和某些专用的规范(spec)结合在一起，这些专用的规范是为创建新的 URL 而提供的。例如，如果游戏是从 `http://croftsoft.com/arcade/` 上下载的，则为 `fileContentsSpec` 提供了一个值 `data`，那么最终结果文件内容的 URL 就会为 `http://croftsoft.com/arcade/data`。

```
public void saveBytesUsingPersistenceService (
    String    fileContentsSpec,
    byte [ ] bytes )
    throws IOException, UnsupportedOperationException
    //////////////////////////////////////
{
    BufferedOutputStream bufferedOutputStream = null;

    try
    {
        PersistenceService persistenceService = ( PersistenceService )
            ServiceManager.lookup ( "javax.jnlp.PersistenceService" );

        URL fileContentsURL
            = createFileContentsURL ( fileContentsSpec );

        try
        {
            persistenceService.delete ( fileContentsURL );
        }
        catch ( FileNotFoundException ex )
        {
        }

        persistenceService.create ( fileContentsURL, bytes.length );

        FileContents fileContents
            = persistenceService.get ( fileContentsURL );

        bufferedOutputStream = new BufferedOutputStream (
            fileContents.getOutputStream ( true ) );

        bufferedOutputStream.write ( bytes );
    }
    catch ( UnavailableServiceException ex )
    {
        throw ( UnsupportedOperationException )
            new UnsupportedOperationException ( ).initCause ( ex );
    }
    finally
    {
        if ( bufferedOutputStream != null )
        {
            bufferedOutputStream.close ( );
        }
    }
}
```

在同一个类中，方法 `saveBytesUsingPersistenceService()` 使用了方法 `createFileContentsURL()`。这个方法没有使用文件名称参数，而是带有一个 `fileContentsSpec` 参数，该参数的作用与文件名称参数相同，只是没有指定一个具体的硬盘驱动器的名称。我常常就使用游戏的名称作为 `fileContentsSpec` 的值，例如 `tank`。

获得 `PersistenceService` 的一个引用以后，这个方法就开始删除与 `fileContentsURL` 关联的以前的数据——如果有这种历史数据的话。然后它就请求一个与要存储字节数组大小正好相等的一个新的存储空间。

注意，该代码使用了可选包 `javax.jnlp` 中的 `FileContents` 的一个实例，而不是标准内核包 `java.io` 中 `File` 类的一个实例。由于该代码使用的是可选包 `javax.jnlp`，该包在您的开发环境中可能存在，也可能不存在，所以 `javax.jnlp` 包中 `UnavailableServiceException` 被包装在标准内核包 `java.lang` 的 `UnsupportedOperationException` 中。

```
public static void saveBytesUsingPersistenceService (
    String fileContentsSpec,
    byte [ ] bytes )
    throws IOException, UnsupportedOperationException
    //////////////////////////////////////////
{
    check ( );

    JNLP_SERVICES.saveBytesUsingPersistenceService (
        fileContentsSpec, bytes );
}
```

正如在前一章中介绍的一样，一般都使用动态类加载和可定制接口来将 `JnlpServicesImpl` 类——这个类包含与可选包 `javax.jnlp` 的静态链接——从其余的代码中隔离开来，而其余的代码不包含与可选包 `javax.jnlp` 的静态链接。为了使游戏既能够在 JNLP 环境中加载也能够在非 JNLP 环境中加载，需要通过调用与上面给出的方法具有相同签名的 `JnlpLib` 中的那个方法，间接访问 `JnlpServicesImpl` `saveBytesUsingPersistenceService()` 方法。如果在当前环境中，JNLP 或 JNLP 的 `PersistenceService` 不可用，该代理方法就会抛出 `UnsupportedOperationException` 异常。

```
public void saveSerializableUsingPersistenceService (
    String fileContentsSpec,
    Serializable serializable )
    throws IOException, UnsupportedOperationException
    //////////////////////////////////////////
{
    saveBytesUsingPersistenceService (
        fileContentsSpec,
        SerializableLib.compress ( serializable ) );
}
```

为了保存 `Serializable` 对象而不是保存字节数组，请使用 `saveSerializableUsingPersistenceService()` 方法。注意，该方法首先会使用 `SerializableLib.compress()` 方法，将 `Serializable` 对象转换为一组字节。为了请求存储空间准确的容量(`PersistenceService` 为存储 `Serializable` 数据对象而需要的存储容量)，必须知道字节数组的长度，因此我们要采用这一中间步骤。


```

public Serializable loadSerializableUsingPersistenceService (
    String fileContentsSpec )
    throws ClassNotFoundException, IOException,
        UnsupportedOperationException
    //////////////////////////////////////
{
    try
    {
        PersistenceService persistenceService = ( PersistenceService )
            ServiceManager.lookup ( "javax.jnlp.PersistenceService" );

        FileContents fileContents = persistenceService.get (
            createFileContentsURL ( fileContentsSpec ) );

        return SerializableLib.load ( fileContents.getInputStream ( ) );
    }
    catch ( FileNotFoundException ex )
    {
        return null;
    }
    catch ( UnavailableServiceException ex )
    {
        throw ( UnsupportedOperationException )
            new UnsupportedOperationException ( ).initCause ( ex );
    }
}

```

上面是 `JnlpServicesImpl` 类中相应的 `load()` 方法。它在 `JnlpLib` 类中也有一个具有相同签名的代理方法。

```

public byte [ ] loadBytesUsingPersistenceService (
    String fileContentsSpec )
    throws IOException, UnsupportedOperationException

```

假定 JNLP 客户端通常可以为您的游戏保证至少有 128K 字节的数据存储空间，那么您可以随意一些，以压缩串行对象的方式存储游戏的数据，而不需要为字节计数的事情而操心。如果您担心会接近或超过这个最小存储空间(128K)，就需要开始非常谨慎地跟踪数据的计数。在这种情况下，可以使用 `loadBytesUsingPersistenceService()` 方法。然后将 `ByteArrayInputStream` 包装成一个 `DataInputStream`，以解析在字节数组内已知索引位置上的 `Object` 和基本类型的数据。

6.2.4 applet 持久性

过去，applet 的开发人员经常会使用静态类变量在 applet 实例之间共享数据。这样，即使 applet 的实例已经销毁，applet 类和静态变量也会在内存中保存，所以使它们能够在超出单个 applet 的生命周期以后，还可以临时地拥有持久性数据。在用户重新加载 applet 的时候，新的 applet 实例会读取由以前的实例创建的静态变量的值。

Java 1.4 在 `java.applet.AppletContext` 类中引入了一些新的方法，这些方法被称为 Applet Persistence API。

这些新的方法使 applet 的开发人员能够从浏览器会话中流出数据和对象，以让这些数据和对象在后续的浏览器会话中能够得到重用。这种方法为 applet 提供了持久性机制，也使在 applet 中使用静态对象来提供持久性数据的这种机制变得不再必要。

——*Java Plug-in 1.4 Developer Guide*⁴

上面引用的这个文档似乎是不正确的，或尚未实现的，或者是有些迷惑人。只有类是存储在内存中，静态对象才可以具有持久性。在浏览器会话结束的时候，JVM 实例也就结束了，所有的数据(包括静态变量)也都会丢失。我的经验告诉我，Applet Persistence API 方法的行为类似，数据会在浏览器会话之间丢失。但是，这些方法对临时地保存和共享数据而言还是很有用的。这就意味着，applet 实例在游戏正在结束的时候，可以保存最高得分。如果 applet 稍后在相同的浏览器会话中重新加载，它就能获得这个数据。如果加载多个 applet 实例，它们就能共享最高得分数据。

类 `java.applet.AppletContext` 现在专门为此提供了两个新方法：`setStream(key, inputStream)` 和 `getStream(key)`。这些新方法允许以由任意一个 `String` 引用的流形式存储和检索数据。我喜欢使用与接口 `java.util.Map` 的 `put()` 和 `get()` 方法相似的思维方式来考虑这个问题，在接口 `java.util.Map` 中存储的是 `Object` 键值对。而在此存储的是一个 `String` 键和一个 `InputStream` 值。

```
public static void saveSerializableUsingAppletPersistence (
    Applet      applet,
    String      key,
    Serializable serializable )
    throws IOException, UnsupportedOperationException
    //////////////////////////////////////
{
    NullPointerException.check ( applet, "null applet" );

    NullPointerException.check ( key, "null key" );

    NullPointerException.check ( serializable, "null serializable" );

    AppletContext appletContext = null;

    try
    {
        appletContext = applet.getAppletContext ( );
    }
    catch ( NullPointerException ex )
    {
    }

    if ( appletContext == null )
    {
        throw new UnsupportedOperationException ( "null AppletContext" );
    }
}
```

4 http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/persistence.html


```

InputStream inputStream = new ByteArrayInputStream (
    SerializableLib.compress ( serializable ) );

appletContext.setStream ( key, inputStream );
}

```

这是 `com.croftsoft.core.applet` 包中 `AppletLib` 类的一个静态方法，可以使用这个方法存储 applet 游戏的数据。游戏的数据必须是以 `Serializable` 对象的形式提供的。参数 `key` 可以是任意的 `String` 值。如果该代码不能访问 `AppletContext` 的话，就会抛出 `UnsupportedOperationException` 异常。在考虑到游戏可能既要以基于浏览器 applet 的形式运行又要以单独桌面应用程序的形式运行之前，您可能始终会认为这好像有点不正常。在游戏以单独桌面应用程序的形式运行时，applet 持久性可能不可用。

```

public static Serializable loadSerializableUsingAppletPersistence (
    Applet    applet,
    String    key )
    throws    ClassNotFoundException, IOException,
    UnsupportedOperationException
    //////////////////////////////////////
{
    NullPointerException.check ( applet, "null applet" );

    NullPointerException.check ( key, "null key" );

    AppletContext appletContext = null;

    try
    {
        appletContext = applet.getAppletContext ( );
    }
    catch ( NullPointerException ex )
    {
    }

    if ( appletContext == null )
    {
        throw new UnsupportedOperationException ( "null AppletContext" );
    }

    InputStream inputStream = appletContext.getStream ( key );

    if ( inputStream == null )
    {
        return null;
    }

    return SerializableLib.load ( inputStream );
}

```

这里是在同一个类中使用同一个键值检索数据的相对应的一个静态方法。

```

GameData gameData = ( GameData )
    AppletLib.loadSerializableUsingAppletPersistence (
        this, PERSISTENCE_KEY );

```

```
highScore = gameData.getHighScore ( );
```

该方法返回的时候，需要将这个 **Serializable** 对象转换为其原始的类，以便访问数据。

使用这 3 个方法的时候，一定要记住，无论是什么名称，新 **Applet Persistence API** 的方法永远都不会超过浏览器 **JVM** 实例的生命周期去维持数据的持久性。如果这在将来也不会改变的话，可能在最新的 **applet** 容器的实现中，会需要使用有符号的 **applet** 或服务端端的代码来真正地持久存储数据。

6.2.5 稳固持久性

您可能想让游戏部署为桌面应用程序、**Java Web Start** 应用程序和 **applet**。这时，您可能希望代码可以使用在它当前所在的环境中可以使用的任意一种持久性机制。

```

public static boolean save (
    Serializable serializable,
    String         latestFilename,
    String         backupFilename,
    String         fileContentsSpec,
    Applet         applet,
    String         persistenceKey )
    throws IOException
    //////////////////////////////////////
{
    NullPointerException.check ( serializable );

    if ( latestFilename != null )
    {
        try
        {
            String userHomeDir = System.getProperty ( PROPERTY_USER_HOME );

            String latestPath
                = userHomeDir + File.separator + latestFilename;
            if ( backupFilename != null )
            {
                String backupPath
                    = userHomeDir + File.separator + backupFilename;

                save ( serializable, latestPath, backupPath );
            }
            else
            {
                save ( serializable, latestPath );
            }
        }

        return true;
    }
}

```



```

    }
    catch ( SecurityException ex )
    {
    }
}

if ( fileContentsSpec != null )
{
    try
    {
        JnlpLib.saveSerializableUsingPersistenceService (
            fileContentsSpec, serializable );

        return true;
    }
    catch ( UnsupportedOperationException ex )
    {
    }
}

if ( ( applet != null )
    && ( persistenceKey != null ) )
{
    try
    {
        AppletLib.saveSerializableUsingAppletPersistence (
            applet, persistenceKey, serializable );

        return true;
    }
    catch ( UnsupportedOperationException ex )
    {
    }
}

return false;
}

```

上面给出的 `SerializableLib` 中的 `save()` 方法将会试图使用 3 种不同的机制持久化一个 `Serializable` 对象。它首先尝试以用户主目录中文件的形式保存该对象。如果游戏部署为无符号的 Java Web Start 应用程序或部署为基于 Web 的 applet，这种尝试就会失败，并抛出 `SecurityException` 异常。然后 `save()` 方法就尝试使用 JNLP 持久性机制保存数据。如果游戏不是部署为使用 Java Web Start 或一些其他的 JNLP 客户端，那么这种尝试也会失败，并同时抛出 `UnsupportedOperationException` 异常。它最后尝试使用 applet 持久性，如果这最后一种尝试也失败了，那么就会返回 `false`。注意，通过这个便利方法，为适当的参数提供一个空值，完全可以不使用上面任何一个持久性机制。

```

public static Serializable load (
    String      primaryFilename,
    String      fallbackFilename,

```

```

String      fileContentsSpec,
Applet      applet,
String      persistenceKey,
ClassLoader classLoader,
String      resourcePathFilename )
throws ClassNotFoundException, IOException
////////////////////////////////////
{
    Serializable serializable = null;

    if ( primaryFilename != null )
    {
        try
        {
            String userHomeDir = System.getProperty ( PROPERTY_USER_HOME );

            String primaryPath
                = userHomeDir + File.separator + primaryFilename;

            if ( fallbackFilename != null )
            {
                String fallbackPath
                    = userHomeDir + File.separator + fallbackFilename;

                serializable
                    = ( Serializable ) load ( primaryPath, fallbackPath );
            }
            else
            {
                serializable = ( Serializable ) load ( primaryPath );
            }
        }
        catch ( FileNotFoundException ex )
        {
        }
        catch ( SecurityException ex )
        {
        }
    }

    if ( ( serializable == null )
        && ( fileContentsSpec != null ) )
    {
        try
        {
            serializable = ( Serializable )
                Jnlplib.loadSerializableUsingPersistenceService (
                    fileContentsSpec );
        }
        catch ( UnsupportedOperationException ex )
        {
        }
    }
}

```



```

    }
}

if ( ( serializable == null )
    && ( applet != null )
    && ( persistenceKey != null ) )
{
    try
    {
        serializable = ( Serializable )
            AppletLib.loadSerializableUsingAppletPersistence (
                applet, persistenceKey );
    }
    catch ( UnsupportedOperationException ex )
    {
    }
}

if ( ( serializable == null )
    && ( classLoader != null )
    && ( resourcePathFilename != null ) )
{
    serializable
        = ( Serializable ) load ( classLoader, resourcePathFilename );
}

return serializable;
}

```

相应的 load()方法也会尝试使用 save()中的这 3 种持久性机制，但是它还增加了第 4 个选项。它假定如果不能从硬盘驱动器、JNLP muffin 或 applet 上下文中加载数据文件的话，那么可能就会是没有以前保存的用户数据。load()方法就再尝试从 JAR 文件中加载固定的初始化数据。

使用这 3 种方法的示例程序就是 com.croftsoft.apps.tile 包中的类 Tile，在前面瓦片地图图像这一节我们曾介绍过这个示例。这个程序加载并显示一个虚拟地图，这个虚拟地图是由代表一些不同类型地形(例如草地、水、树等)的正方形瓦片组成。用户通过简单地单击目标正方形，就可以改变地面的类型。当用户退出瓦片编辑器的时候，任何一种持久性机制只要可用，这种更改就会保存下来。

```

public void init ( )
///////////////////////////////////////////////////
{
    super.init ( );

    TileData tileData = null;

    try
    {
        tileData = ( TileData ) SerializableLib.load (
            LATEST_FILENAME,
            BACKUP_FILENAME,

```

```

        FILE_CONTENTS_SPEC,
        ( Applet ) this,
        PERSISTENCE_KEY,
        getClass ( ).getClassLoader ( ),
        RESOURCE_PATH_FILENAME );
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }

    if ( tileData == null )
    {
        try
        {
            tileData = TileData.loadTileDataFromImage (
                TILE_MAP_IMAGE_FILENAME,
                getClass ( ).getClassLoader ( ) );
        }
        catch ( IOException ex )
        {
            ex.printStackTrace ( );
        }
    }

    if ( tileData != null )
    {
        palette = tileData.getPalette ( );

        tileMap = tileData.getTileMap ( );
    }

    try
    {
        TileData.remapToPalette ( palette, tileMap, DEFAULT_PALETTE_INDEX );
    }
    catch ( IllegalArgumentException ex )
    {
        palette = new int [ ] {
            0xFF0000FF,    // blue water
            0xFF00FF00 }; // green land

        tileMap = TileData.generateRandomTileMap (
            new Random ( RANDOM_SEED ),
            palette,
            DEFAULT_MAP_SIZE, // rows
            DEFAULT_MAP_SIZE, // columns
            SMOOTHING_LOOPS );
    }

    [...]
}

```


上面的示例代码是从 `Tile.init()` 中抽取出来的。每一次启动 `Tile` 程序的时候，它就尝试从用户主目录文件、JNLP muffin、applet 上下文、JAR 文件和瓦片地图图形中加载地形，而且就按上面给出的顺序进行尝试。如果所有的这 5 种加载机制都失败，或加载的数据被破坏，那么 `Tile` 最后就会产生随机的地形数据。

```
public void destroy ()
///////////////////////////////////////////////////
{
    if ( dataIsDirty )
    {
        try
        {
            SerializableLib.save (
                new TileData ( palette, tileMap ),
                LATEST_FILENAME,
                BACKUP_FILENAME,
                FILE_CONTENTS_SPEC,
                ( Applet ) this, PERSISTENCE_KEY );
        }
        catch ( Exception ex )
        {
            ex.printStackTrace ( );
        }
    }

    super.destroy ( );
}
```

如果地形数据已经被更改，那么方法 `destroy()` 就会使用任意一种可用的持久性机制来存储地形数据。该方法首先检查布尔标志 `dataIsDirty`，以便检查这种保存是否必要。只要用户通过单击的方式更改了任何一个瓦片，那么程序就会设置 `dataIsDirty` 标志。

正如在 `Tile` 示例中给出的一样，使用 `SerializableLib` 中的便利方法 `load()` 和 `save()` 可以充分降低定制代码的数量——定制代码是在游戏中为了实现基本的持久性而编写的代码。大部分工作都将会朝着创建 `Serializable` 数据对象类的方向努力，这种 `Serializable` 数据对象类对每一个应用程序而言可能都是特别具体的，而且在整个开发过程中，可能会频繁地进行更改。

当然，还有很多其他的持久性机制。虽然我不像喜欢这里给出的这些方法一样喜欢它们，但是还是有一些需要它们使用的环境。其中的几种机制以及它们与游戏编程的关系将在下面进行讨论。

6.2.6 嵌入式数据库

提到数据的时候，您马上就会想到的一种持久性机制，那就是数据库。虽然，指望游戏玩家事先安装一个游戏可以使用的数据库管理系统(DBMS)可能有些过分，但是将 DBMS 作为安装程序的一部分倒是非常可行的。可以在游戏中嵌入一个纯 Java 的轻量级的关系 DBMS(RDBMS)或面向对象的 DBMS (OODBMS)。如果选择使用 RDBMS，就可以通过使用内核包 `java.sql`，使用结构化查询语言(Structured Query Language, SQL)和 Java 数据库连接(Java Database Connectivity, JDBC)的 API 访问数据。

但是,在我看来,在游戏中加载一个嵌入式数据库系统所需许可的费用可能比较昂贵。数据库对多个用户需要并发读写访问数据的应用程序来说,可能是最好的一种选择。另一方面,客户端的游戏一般都只有一个玩家,而且数据的完整性、数据报告和分析需求的要求较低。

除了并发访问以外,会考虑使用 DBMS 的原因之一可能就是不需要同时将所有的数据都加载到内存。嵌入式数据库,尤其是 OODBMS,对查询那些在任意给定的时间都可能需要的数据并将它们加载到内存而言,可能会是一个快速和便利的方法。相反,使用串行化的对象或 XML 编码的对象可能需要将一整个图形对象都加载到内存,才能检索到其中需要的一个数据对象。

在运行到内存越界的时候,就会明白遇到了这种事情。遇到这事情的另外一个标志就是,要花很长的时间才能将数据从硬盘加载到内存,因为要将所有的数据都加载内存,而实际上在某一个给定的时刻,可能仅仅需要几个字节的数据。这种情况的另一个示例可能就是在启动初始化的过程中,仅需要将游戏第一关的数据加载到内存,而不是将游戏所有关的数据都加载到内存。当然,通过将游戏每一关的数据分别存储在一个单独的文件中,也可以解决这个问题。一个更好示例可能就是仅为虚拟游戏世界中直接接壤的一个任意视图加载地形和实物对象的数据。

在这种情况下,我仍然会使用一个随机访问数据文件而不使用数据库,来回避这个问题。在我看来,绝大部分游戏可能都不会需要像嵌入式数据库这样的怪异的东西。但是,在不远的将来,我可能会改变看法,因为 Java 数据对象(Java Data Objects, JDO)API 的开放源代码(Open Source)实现已经初露端倪。

6.2.7 服务器端的持久性

如果因为安全沙箱的限制而不能在用户的客户机上保存数据,那么您可能会想在服务器上保存那些数据。我们知道,applet 并没有限制向它的代码库服务器建立一个 Internet 链接。代码库服务器可以存储或传播上载到它上面的任何用户存储的数据。服务器端持久性的优势是用户可以从任一台与 Internet 链接的计算机上继续玩以前保存的游戏。缺点是现在需要支持服务器端的代码。如果游戏是集中式的,或半集中式多玩家联网式的,那么就总会需要这种基本结构。

我的建议是,应该启动较小的服务器端代码,但是要用游戏可以很容易扩展的一种方式编写服务器端代码(当游戏流行起来以后)。对多玩家在线游戏而言,可以从一开始就使用 J2EE 的 Enterprise Java Beans(EJB)来保证其可升级性。可以通过使用 Open Source EJB 应用程序服务器启动较小的服务器端代码。

我推荐使用的应用程序服务器是 Open Source 的产品 JBoss⁵。正如您对 Open Source 产品的期望一样,JBoss 也是一个免费产品,但是这并不是说该产品就比商业化的产品差。据说在产品性能方面,它实际上比其他的一些竞争产品都更加优秀。另外,您可以找到 Web 主机服务,只需要付很少的费用就可以获得在启动程序时需要的 Jboss 支持。例如,WebAppCabaret 就提供了含有 Jboss 支持的一个包,而每月的费用少于 30 美元⁶。

我过去习惯推荐使用 Open Source servlet 容器 Tomcat 和简化的 SerializableLib 对象串行化机制来获取基本的服务器端持久性机制,并让这种服务器端持久性机制快速运行。后来,在设

⁵ <http://www.jboss.org>

⁶ <http://www.webappcabaret.com>

计确定下来以后，我又绕了回来，转向 JDBC 关系数据库的持久性的机制。我更愿意使用数据访问对象(data access object, DAO)接口来屏蔽其余的服务器端代码，使其不发生持久性机制的切换。

```
package com.croftsoft.apps.wyrm.entity;

[...]

import javax.ejb.*;

[...]
public abstract class PcBean
    implements EntityBean
    //////////////////////////////////////
    //////////////////////////////////////
    {

    public abstract Long    getId ( );

    public abstract String getName ( );

    public abstract String getState ( );

    public abstract long    getHealth ( );

    public abstract long    getWealth ( );

    public abstract long    getLevel ( );

    public abstract long    getExperience ( );

    //

    public abstract void setId      ( Long id      );

    public abstract void setName    ( String name  );

    public abstract void setState   ( String state );

    public abstract void setHealth  ( long health  );

    public abstract void setWealth  ( long wealth  );

    public abstract void setLevel   ( long level   );

    public abstract void setExperience ( long experience );

    [...]

    //////////////////////////////////////
    //////////////////////////////////////
    }
```

但是, 现在, 我建议您一开始就使用 EJB 实体 bean 的持久性机制。我改变思路的原因是, 现在有了 EJB 容器管理的持久性(Container Managed Persistence, CMP)。为抽象类提供数据对象的属性(该属性是由抽象 accessor 方法和 mutator 方法的签名描述的)时, 应用程序服务器的 CMP 特性就会自动为您产生所需要的数据库表和数据库查询。我已经使用 JBoss 中的 CMP 实现和与之共存的默认 Open Source DBMS, 成功地测试了上面给出的 PC 实体 bean 代码。我也成功实现了 JBoss 和 Oracle DBMS 的联合。由于具体使用的 DBMS 是从 EJB 标准中抽象出来的, 所以可以从一种 DBMS 实现切换到另一种 DBMS 实现, 而不用改变游戏的代码。

6.3 数据完整性

您可能希望能够有一个在玩家的客户机器上存储玩家数据的网络游戏——即使在这个游戏网络环路中存在一个服务器也是如此。可能是这种情况: 您只想让服务器成为多个游戏玩家之间的一个消息中继器(message relay)。另外一个可能就是您有一个单人模式的游戏(或多玩家模式的游戏), 而只想让服务器充当一个裁判的角色, 处理玩家客户机的请求和产生结果, 而不会存在任何欺骗——因为服务器代码是可以信赖的。由于花费或责任的原因, 您可能不想为服务器端玩家的数据持久性提供基础构造。

在客户机上存储玩家数据带来的问题是有些玩家为了实现欺骗而会专门攻击他人的数据。虽然没有什么办法防止他们在网络客户端上为所欲为, 但是您还是可以可靠地检测到他们个人游戏数据中的某些坏数据。而此时, 内核包 java.security 中的一些加密类有了用武之地。

6.3.1 消息摘要

一般可以使用 MessageDigest 类存储不限长度的有序字节序列的一个惟一标识符。如果某个文件中有一个字节发生了变化, 它的数字指纹就会完全发生变化。可以将一个消息摘要作为一个校验位或几乎不可能进行欺骗的循环冗余校验(CRC)值。

当您将一个数据文件从服务器发送到客户机进行持久存储时, 就可以在服务器上将消息摘要归档。如果采用的是安全散列算法(Secure Hashing Algorithm, SHA), 那么为玩家存储的消息摘要的长度都将会是 20 个字节, 无论数据文件的长度是多少。当玩家想要重新开始游戏时, 客户软件就会将保存的游戏数据上传到服务器。在服务器上, 以前的消息摘要和新计算出来的值会进行一个快速的比较, 就能确定数据文件是否被攻击过或被破坏过。

```
public static byte [ ] digest (
    InputStream inputStream,
    String      algorithm )
    throws IOException, NoSuchAlgorithmException
    //////////////////////////////////////
{
    BufferedInputStream bufferedInputStream
        = new BufferedInputStream ( inputStream );

    MessageDigest messageDigest
        = MessageDigest.getInstance ( algorithm );

    int i;
```



```

while ( ( i = bufferedInputStream.read ( ) ) > -1 )
{
    messageDigest.update ( ( byte ) i );
}

bufferedInputStream.close ( );

return messageDigest.digest ( );
}

public static byte [ ] digest (
    File    file,
    String algorithm )
    throws IOException, NoSuchAlgorithmException
    //////////////////////////////////////
{
    return digest ( new FileInputStream ( file ), algorithm );
}

public static byte [ ] digest ( File file )
    throws IOException, NoSuchAlgorithmException
    //////////////////////////////////////
{
    return digest ( file, "SHA-1" );
}

```

如上所示，包 `com.croftsoft.core.security` 的 `DigestLib` 类中包含用于这种目的的一些很有用的方法。但是要记住，为了保证数据的完整性，该操作必须执行服务器端那些在服务器上以独占方式进行更新的数据上。您必须始终假定从客户端传来的数据都是值得怀疑且不可信赖的，因为客户软件是可以被攻击的，或网络消息是可以被欺骗或替换的。

如果您赞成这种观点，那么可能就会希望将数据进行数字签名。使用这种加密算法，可以检验由服务器产生的以前保存的那些数据文件，并可以检验这些文件是否被更改过，并且不需要保存用来作比较的消息摘要。在数字签名技术方面，我并没有什么经验或代码可以与您进行分享，但是我知道这种技术在核心 `Java` 编程语言里得到了很好的支持。更多的信息可以参考包 `java.security` 中的 `Security` 类。

6.3.2 散列缓存

我常常要花很多时间来开发持久性的缓存机制。这样做的问题是，随着开发过程的继续，`Java` 游戏 applet 和桌面应用程序会成倍增长，并需要频繁地进行更新、递增。通过仅在需要的时候和所需要的数据发生变化时，才下载那些所需要数据的解决办法，那些含有很多代码和多媒体资源的高质量游戏，就可以在那些使用拨号上网的计算机上运行。

既然可以使用 `Java Web Start` 和 `Java` 插件 applet 缓存技术，那么我也就不再担心这一点。如果您想看看我过去都做了些什么，可以浏览一下 `com.croftsoft.core.lang.classloader` 包中的一些源代码文件。基本的思想是，实现一个可定制的 `ClassLoader` 子类，该子类会按照需要从网络上获取类文件和多媒体文件，并将它们持久地缓存到硬盘中，以备后用。

我提到这个现在已经很陈旧技术的原因是，确实有这种技术的一个组件，这个组件在某些环境中仍然十分有用。定制的 `ClassLoader` 去做的一个事情就是，在服务器上有新类和多媒体文件的时候，它会自动并递增地更新缓存的这些类和多媒体文件。启动游戏的时候，它就会从服务器上下载一个清单，该清单包含一些由应用程序使用的全部编译过的文件和资源文件的最新消息摘要。当游戏请求 `ClassLoader` 将类、图像或音频文件加载到内存的时候，它首先就会检查客户机上缓存备份的摘要是否与服务器上最新版本的摘要信息匹配。如果不匹配，就会下载新版的摘要信息，也会更新持久性磁盘的缓存。

这对递增的更新来说非常重要，尤其是在开发过程中频繁进行修改的时候。我也包含了一个可选项，在游戏启动的时候预先验证所有的缓存文件，这样任何网络延迟都不会中断后面的游戏。这有助于减少由用户偶然破坏他们的游戏应用程序文件而带来的技术支持方面的请求。消息摘要的改变会被检测，被更改的文件也会被替换掉。

最终，这种技术可以用来创建一个持久性缓存，该缓存可以由多个应用程序共享。由一个游戏下载的类或资源文件可以从缓存中保存下来，并由另一个游戏使用。即使最开始下载和缓存该文件的程序是不可信赖的，这种功能也一样有效。即使为该文件提供服务的下载服务器是不可信赖的，这种功能还能照常有效。只要知道缓存的文件与想要的文件是严格逐位匹配的，就像由消息摘要的检查保证的一样，缓存文件的源就无关紧要。

文件是否是经过认证的——无论是数字签名或是某些认证授权——已经无关紧要。如果可以确定文件精确地与游戏的需要(基于清单中列出的一个摘要所涵盖的需求)相匹配，就不但可以不去关心它是从哪里来的，而且也不用关心是谁最初产生这个数据文件的。

这使我非常兴奋，因为这又重新让开发人员去决定使用哪一个库。例如，您希望选用一个更好的 GUI 组件库，也许就是一个 `Open Source`。但是，又不愿意部署它，因为它不是预安装的，在游戏下载中，必须将上千兆的字节打包。然后，假想您运行一些测试程序，并很高兴地发现，虽然 JVM 或浏览器(大部分客户机都使用 JVM 或浏览器)的供应商并没有预安装这些库文件，但是它实际上是已经安装上的，并且在共享的持久性缓存内部，大部分客户机都可以访问这些库文件。库越是受到开发者的欢迎，它就越会被缓存。它越被缓存，开发人员就越会欢迎。

这种思路也适合于资源文件，例如公共领域中的精灵图像库和音频库。无论文件是什么类型的，随着共享缓存的出现，供应商统治和预安装能力变得也越来越无关紧要，因为不存在发布的垄断性。我企盼 `Public Domain` 和 `Open Source` 共同支配客户缓存。有文件要发布的开发人员(这些文件是从原来旧文件中改进的，并且是按照像效率或质量这样的标准进行标准化的)，可以直接从他们的 Web 站点上启动发布，有没有 JVM 或浏览器供应商的支持都无所谓。也不会劝阻其他开发人员采用基于发布的新库。

我在散列缓存上进行了一些研究，并开发了一个专供开发使用的原型系统。我也想在这方面取得一个专利，直到发现在先技术(prior art)以后，才放弃这个想法。如果您愿意使用我的散列缓存原型(这样命名是因为在消息摘要的产生过程中，使用了安全散列算法)，那么请参考 `com.croftsoft.core.util.cache.secure` 包中的 `Open Source` 类。那些文件，包括我已经丢弃的专利申请书，都是可以使用的——通过 CroftSoft Tutorials 页面⁷上的 Hash Cache 链接。这种没有经过鉴定的安全共享缓存技术最终必然会并入大部分 Web 浏览器和 JNLP 客户机的方法，这也是我所希望的。

7 <http://www.croftsoft.com/library/tutorials/>

6.4 小结

我强烈推荐您使用 `SerializableLib` 中具有“稳固持久性”的 `save()` 和 `load()` 这两个非常好用的方法，将您在持久性数据方面的工作量降到最小。除非您有更合理的理由选择其他方法，否则可以坚持使用对象串行化来保存和加载游戏数据。这样做的效果一般都很好。如果觉得不够好，可能仍然会启动 `SerializableLib` 方法作为最初客户端持久性数据的实现，并进行调整以获得更好的效率，直到游戏其余的部分全部完成。记住，通常都希望能够尽快地获得这种基础构造，并继续进行有意义的程序设计。

6.5 参考文献

Croft, David Wallace. “Hash Cache,” 2000-08-19. <http://www.croftsoft.com/library/tutorials/>.

Marinescu, Floyd. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. Hoboken, NJ: John Wiley & Sons, 2002.

McLaughlin, Brett. *Java & XML*, 2nd Edition. Sebastopol, CA: O'Reilly & Associates, 2001.

Monson-Haefel, Richard. *Enterprise JavaBeans*, 3rd Edition. Sebastopol, CA: O'Reilly & Associates, 2001.

Sperberg-McQueen, C.M. and Henry Thompson. “W3C XML Schema,” 2003-01-01. <http://www.w3.org/XML/Schema>.

Van Haecke, Bernard. *JDBC 3.0: Java Database Connectivity*. New York, NY: M&T Books, 2002.

第 7 章 游戏体系结构

及时的一针胜过事后的九针。

——本杰明·富兰克林

“游戏体系结构”这个术语应该与“游戏设计”这个术语进行区分。游戏设计是指游戏看起来大概像个什么样子和该游戏是怎样玩的。而另一方面，游戏体系结构是指游戏是怎样构建起来的。一般而言，在软件这个行业中，术语“设计”通常指的就是这本书中称为“体系结构”的东西。但是在游戏开发行业中，游戏设计者创建游戏设计而游戏程序员创建游戏的是游戏的体系结构。

除了从性能方面考虑的可能因素以外，游戏的体系结构一般是不会显示给游戏玩家的。但是，它也是非常重要的东西，因为底层体系结构的选取对完成游戏的能力和及时发布修订版的能力而言是非常重要的。本书的前面几章主要关注于可重用游戏库组件。这一章主要介绍如何将这组件组装成一个完整的游戏。最后，还将介绍数据驱动模式设计的优点。

7.1 接口

如果想构建一些小游戏，可能不需要对体系结构考虑很多。我们可能会相当快速地使用最小量的代码，编写很多个较简单的街机游戏。然而，对那些复杂的游戏来说，如果想尽量降低开发过程中重新设计的工作量，可能就需要对必需的对象以及它们之间的交互关系进行考虑。这个道理是从大家都熟悉的一件事情——建造一个小房子与建造一个摩天大楼中得出的。对建造一个摩天大楼来说，在开始建造以前，您可能需要一套设计蓝图。

但是，就基于组件的软件设计来说，摩天大楼的类比也有它的局限性。更好的类比可能是建造一个组合式太空站。如果在太空中有一个悬浮的建筑物，就可以选择在该建筑物上添加新的结构部分。您可以先建造一个小房子，然后逐渐建成摩天大楼，而在这个过程中，不必更换您的建造物的地基。

关于太空站，您最关心的是将来的可扩展性和灵活性。您应该在需要附加功能的时候，可以加挂新的模块；在原来的模块功能失效以后，可以将它们替换掉。由于模块是在它们的接口上相互绑缚在一起的，所以您可以很容易地用更新以后的模块替换原来的模块——只要它们的接口是相互兼容的。当开始构建游戏的时候，您就会希望使用接口代替抽象类或具体类来连接模块。一般做法是，使用接口引用代替具体类的引用，可以提高体系结构的灵活性和组件的可重用性。

7.2 继承

作为一种面向对象的语言，Java 当然也支持继承。您可以创建由游戏中全部对象共享的一个通

用类，该通用类提供游戏引擎所需要的基本功能和兼容性。具体的逻辑是通过从基类中扩展进行处理的。作为这方面的一个简单示例，您可以参考 `Zombie` 类，该类扩展了基类 `Thing`。一般的游戏引擎可以很快地在一组 `Thing` 实例上进行迭代循环，来更新它们(指 `Thing` 实例)在虚拟空间中的位置。一些更具体的逻辑可以确定 `Thing` 的某一个子类使用圣水后的效果。

由于这种方法让人觉得吃惊，所以我的建议是，应该避免从具体类继承。具体类继承的时候，子类也就继承了超类的变量。除非从整体上对对象有一个完全的理解，否则，能够探测到一个对象的内核将是非常危险的。例如，我曾经接过这种具有挑战性的任务：升级由另一程序员编写的一个在线小游戏，而那时这个程序员已经不在这个公司了。编写这个程序的程序员是以十层深的继承链方式构建这个程序的，在这个继承链中，每一个子类的方法都会操纵多个变量，然后再委托给超类的方法进行进一步的处理。不幸的是，这也就意味着在子层被操纵的一个变量，却可能在它的任何一个父层中定义，或被它的任何一个父层重写。安全修改和调试该程序的惟一方法是研究和理解继承链中每一个类的内部工作过程。

但是，严格地通过方法去操纵对象还是相当安全的，因为这样利用了对象封装的好处。不用从超类扩展来继承它的功能，就可以维护指向这种类的实例的一个内部引用。只要需要某个功能，就可以通过对象的方法来访问这个功能。这就是通常所说的黑盒重用，因为对象内部发生的事情是不可见的。上面那种我们禁止采用的方法被称为白盒重用。

```
public final class TileAnimator
    implements ComponentAnimator
    //////////////////////////////////////
    //////////////////////////////////////
    {
private final TilePainter tilePainter;

[...]

public TileAnimator (
    TilePainter tilePainter,
    int          deltaX,
    int          deltaY )
    //////////////////////////////////////
    {
        NullPointerException.check ( this.tilePainter = tilePainter );

        this.deltaX = deltaX;

        this.deltaY = deltaY;

        [...]
    }

[...]

public void update ( JComponent component )
    //////////////////////////////////////
    {

        int offsetX = tilePainter.getOffsetX ( );
```

```

        int offsetY = tilePainter.getOffsetY ( );

        [...]
    }

    public void paint (
        JComponent component,
        Graphics2D graphics )
    {
        ///////////////////////////////////////////////////
        tilePainter.paint ( component, graphics );
    }

```

包 `com.croftsoft.core.animation animator` 中的类 `TileAnimator` 演示了黑盒重用的过程。`TilePainter` 类的方法 `paint()` 绘制一个瓦片网格。`TilePainter` 类的方法 `update()` 驱动这些瓦片产生背景移动的那种幻觉。访问由 `TilePainter` 维护的数据和绘制这些瓦片的能力可以通过具体类的继承来获取。同样地，通过在 `TileAnimator` 内部维护指向 `TilePainter` 委托实例的一个引用，也可以达到相同的目的。`TilePainter` 对象的封装因此得以保护。

这也有助于克服多重继承的问题。Java 不支持来自于具体类的多重继承，因为方法实现的继承是产生混乱的根源，也会降低生产效率。但是，假设需要一个从 `Humanoid`、`Thing` 和 `Undead` 类继承而来的 `Zombie` 类。由于 Java 只支持类的单一继承，所以可能要通过这样的一个继承链：`Zombie`、`Undead`、`Humanoid`、`Thing`，来实现整个目标。但是并不是所有 `Undead` 的实体都是 `Humanoid` 的实体，`VampireBat` 就是这样的一个示例，因此这样尝试：`Zombie`、`Humanoid`、`Undead`、`Thing`。现在 `Humanoid` 的所有子类也都是 `Undead`，这也破坏了多态关系的一致性。

这种问题的解决办法是使用多重接口继承。由于 Java 支持接口的多重继承，所以具体的类 `Zombie` 可以实现接口 `Undead`、`Humanoid` 和 `Thing`。这在解决了多态关系问题的同时，仍然需要继承这些类(实现这些接口的类)的具体功能。通过剪切和粘贴代码，将 `ambulate()` 方法的实现复制到实现了 `Humanoid` 接口的每一个类中，也可以解决这个问题。然后还可以通过使用过程化的编码风格，将 `Undead` 对象的实例传递到一个静态的 `regenerate()` 方法，适当地整理一下代码就可以了。

但是，我推荐的方法是，应该使用接口合成式设计(*design by interface composition*)。`Zombie` 类可以沿用或重写它的 `Humanoid.ambulate()` 和 `Undead.regenerate()` 方法的调用，以委托这些接口具体实现的实例。由于使用的是委托对象，所以可以获取多重继承的多态性优势，而不会产生通常会随之而来的混乱，这还避开了白盒重用的风险。此外，由于 `Zombie` 类以实例变量的形式通过接口引用而不是具体类引用保存委托的对象，所以它从来不知道也不关心具体实现的类到底是什么。这说明合成的对象可以在运行时交换它的行为，按照需要，可以将步法从具体类 `DefaultHumanoid` 的标准步法改变为 `MonstrousHumanoid` 的拖曳步法。

有时我想，如果 Java 编程语言只允许接口继承和对具体对象的接口引用，那么程序员的效率可能就会更高。我想这对可重用 API 而言可能尤为重要。在我看来，最好的一个 API 就是内核包 `java.util` 中的 `Java Collections API`。它是完全基于接口的。另外一个很好的示例是可选包 `javax.jms` 中 J2EE 的 `Java 消息服务(Java Message Service, JMS) API`。它几乎完全是由接口组成的。具体类的实现是由供应商提供的，并且不能通过应用程序的代码直接引用。

几年以前,在我使用早期版本的 Java 3D API 的时候,曾遇到过麻烦,因为它是基于具体类的一个定义好的集合的。API 中的很多方法都被设计为仅接收具体类参数。这说明,传入的所有对象都必须是预期的具体类或子类的一个实例。在有些情形下,就不可能创建一个子类来完成需要完成的任务,要么是因为类被声明为 `final` 类型的,要么就是因为多重继承的问题。这种灵活性的缺乏妨碍了我为一个早期的 bug(该 bug 总是妨碍项目的完成)创建一个工作区的尝试。创建可重用游戏库或引擎 API 的时候,特别是如果计划让它在组织以外的地方可用的话,请记住 API 中的字母 I 在这里表示的是 Interface。

7.3 目标 Mars

建造一个太空站以后,下一个逻辑步骤就是到火星去旅行。这一章其余部分中使用的大部分示例代码都是从我编写的这个游戏(该游戏模拟了火星上的坦克大战)截取下来的(如图 7-1 所示)。您可以在这本书的站点上运行这个游戏,或使用 Ant 的目标 `mars` 构建和启动它。它的源代码可以在 `com.croftsoft.apps.mars` 包中找到。在后面的一些章中,我们将在介绍多玩家联网模式和人工智能时,重新回顾这个游戏。

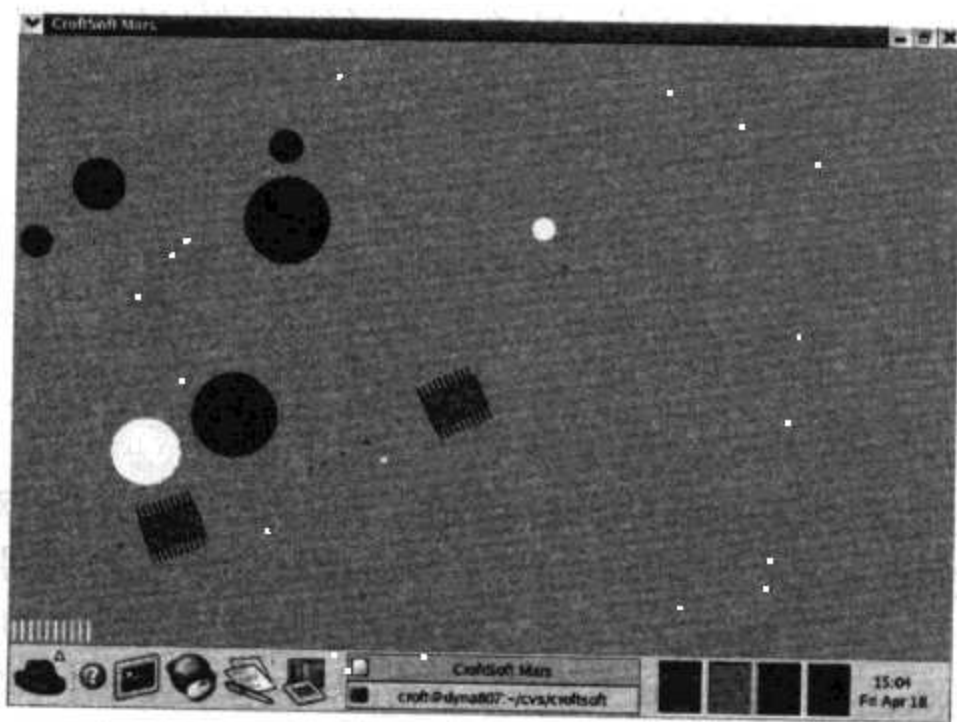


图 7-1 Mars

7.4 模型-视图-控制器

也许您有一个以某个小的类为基础的游戏。在 `init()` 方法中加载所有的多媒体文件和数据;实现 `MouseListener` 方法和 `KeyListener` 方法处理用户的输入;在 `update()` 方法内部更新精灵的位置;在 `paint()` 方法中绘制精灵;在 `destroy()` 方法中保存最高分记录。对那些小游戏而言,这样做很好。

但是,小游戏都会有逐渐增大的趋势,在您清楚这一点以前,那些密集类已经变成了一个难于理解的集成块。您需要把这个类重新分解到多个类中,这样才可以用不同的对象封装数据和相关逻辑,只有如此,才可以理解正在发生的事情。模型-视图-控制器(Model-View-Controller, MVC)就是一种很有名的面向对象的设计模式,在重新分解类的过程

中，可以将这种设计模式作为一个向导使用。MVC 可以帮助您确定怎样将游戏分解成易于管理的大模块，还可以帮助确定这些新对象间的接口应该是什么。

7.4.1 模型

模型对象包含模拟游戏世界中一个实体的某些东西的状态和行为，例如装甲坦克。accessor 方法提供模型当前状态的只读访问，mutator 方法提供更改状态的手段。状态更新可能会限制在模拟行为的一个有限范围以内。如果这样的话，mutator 方法就负责执行这些限制，并将状态更新命令作为一个请求对待。另外，模型可以根据行为的内部规则，更新它自己的状态。

1. ModelAccessor

```
package com.croftsoft.apps.mars.model;

import java.awt.Shape;

public interface ModelAccessor
///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////
{

    public boolean isActive    ( );

    public Shape getShape      ( );

    public boolean isUpdated   ( );

    public double getZ         ( );
```

存取器接口用来提供对象数据的只读访问。通过存取器接口不能修改对象。存取器接口中所有的 accessor 方法返回通过值复制的基本类型、不变的对象引用，或返回辅助的存取器接口。

ModelAccessor 是一个存取器接口，它为游戏中 Model 的状态提供了只读访问。它的 accessor 方法返回 boolean 和 double 基本类型以及指向内核包 java.awt 中存取器接口 Shape 的一个对象引用。方法 isActive() 表示 Model 当前是否正在运行。测试活动状态的指示器有时可能比在游戏结束后就将 Model 从内存中删除更加有效，尤其是可能马上就要重新玩游戏时。getShape() 方法返回 Model 当前的边界，用于检测碰撞和交叉点。这里使用形状而不是体积表示我们正在模拟的是一个 2D 的虚拟世界。方法 isUpdated() 表示在最近的游戏循环迭代中，Model 的状态是否已被更改。方法 getZ() 获取 Model 的 z 轴的坐标，这样背景精灵将会先在前景精灵之前绘制出来。它还可以作为检测碰撞的一个辅助参数，在这种情况下，虚拟世界将会由分层的 2D 实体平面组成。

2. Model

```
package com.croftsoft.apps.mars.model;

public interface Model
    extends ModelAccessor
///////////////////////////////////////////////////////////////////
```



```

////////////////////////////////////
{

    public void setCenter (
        double x,
        double y );

    public void prepare ( );

    public void update ( double timeDelta );
}

```

接口 **Model** 是 Mars 游戏世界中实体的一个基本接口。使用接口而不是使用抽象类或具体类是为了获得灵活性。**Model** 通过添加 **mutator** 方法扩展了接口 **ModelAccessor**。方法 **setCenter()** 在 2D 空间中重新定位实体，在处理过程中更新边界的形状。z 轴方向上没有 **mutator** 方法，因为这个游戏中，我们假定 z 轴方向是固定的(图像是 2D 的)。调用方法 **prepare()** 方法就发出这样的信号：为了做好调用 **Model** 的 **update()** 方法的准备，**Model** 应该重置所有的状态转换标志。使用方法 **update(timeDelta)** 来通知 **Model** 根据它自己行为的内部规则，递增地更新它的模拟状态。参数 **timeDelta** 表示模拟应该提前多少秒。例如在 1/40 秒的时间里，每秒 40 米速度的坦克就会前进一米，而每两秒为它自己补充一颗子弹的临时军火供应站在这段时间里，就会自动补充 0.0125 颗子弹。

3. Damageable

```

package com.croftsoft.apps.mars.model;

public interface Damageable
    extends Model
{
    //////////////////////////////////////
    //////////////////////////////////////

    public void addDamage ( double damage );
}

```

接口 **Damageable** 通过添加 **addDamage()** 方法扩展了 **Model**。并不是模拟的所有游戏实体都是 **Damageable** 的。在游戏世界里，只有这些会遭受破坏和与破坏相关的模型才会实现这个接口。在游戏 Mars 中，这就要包括 **AmmoDump**、**Obstacle** 和 **Tank**，但是不包括 **Bullet**。当 **AmmoDump** 的实例遭受到破坏的时候，它们就会爆炸；当 **Obstacle** 的实例遭受到破坏的时候，就会缩小；当 **Tank** 实例遭受到大量破坏以后，它们就会被销毁。因为 **Damageable** 扩展了 **Model**，所以我们只迭代 **Damageable** 接口的实例，并使用从 **Model** 继承的 **getShape()** 方法来检查和 **Bullet** 位置的交叉点。

4. Impassable

```

package com.croftsoft.apps.mars.model;
public interface Impassable
    extends Model

```

接口 `Impassable` 也扩展了 `Model`。该扩展没有提供辅助方法，但是却指出了该实体将会阻碍游戏中另外一个实体的移动。因为我们正在使用接口，所以可以使用多重继承来标识 `Model` 的子接口(例如 `Tank`)既是 `Damageable` 的又是 `Impassable` 的。

5. AmmoDumpAccessor

```
package com.croftsoft.apps.mars.model;

import java.awt.Shape;

public interface AmmoDumpAccessor
    extends ModelAccessor
    //////////////////////////////////////
    //////////////////////////////////////
{

    public double getAmmo          ( );

    public boolean isExploding     ( );

    public Shape getExplosionShape ( );
```

存取器接口 `AmmoDumpAccessor` 扩展了 `ModelAccessor`，以提供模拟游戏世界里的临时军火供应站实体的一个对象状态的只读访问。坦克没有子弹的时候，就会驶向临时军火供应站去重新加装弹药。当临时军火供应站被击中的时候，就会爆炸。

6. AmmoDump

```
package com.croftsoft.apps.mars.model;

public interface AmmoDump
    extends Model, AmmoDumpAccessor, Damageable
    //////////////////////////////////////
    //////////////////////////////////////
{

    public void setAmmo ( double ammo );
```

接口 `AmmoDump` 扩展了接口 `Model`、`AmmoDumpAccessor` 和 `Damageable`，并提供了一个 `mutator` 方法来设置当前军火弹药的数量。对这种特殊类型的 `Model` 使用的是一个接口而不是具体的类，为的是增大灵活性。

7.4.2 多重接口继承

图 7-2 是一个类图，该类图描绘了游戏中模型接口之间的继承关系。4 个实体类型分别是 `AmmoDump`、`Bullet`、`Obstacle` 和 `Tank`。两个特征类型是 `Damageable` 和 `Impassable`。不要被这些随意的实体类型和特征类型这样的名称所困惑。由于它们都是从 `Model` 继承的，所以它们可以换用。

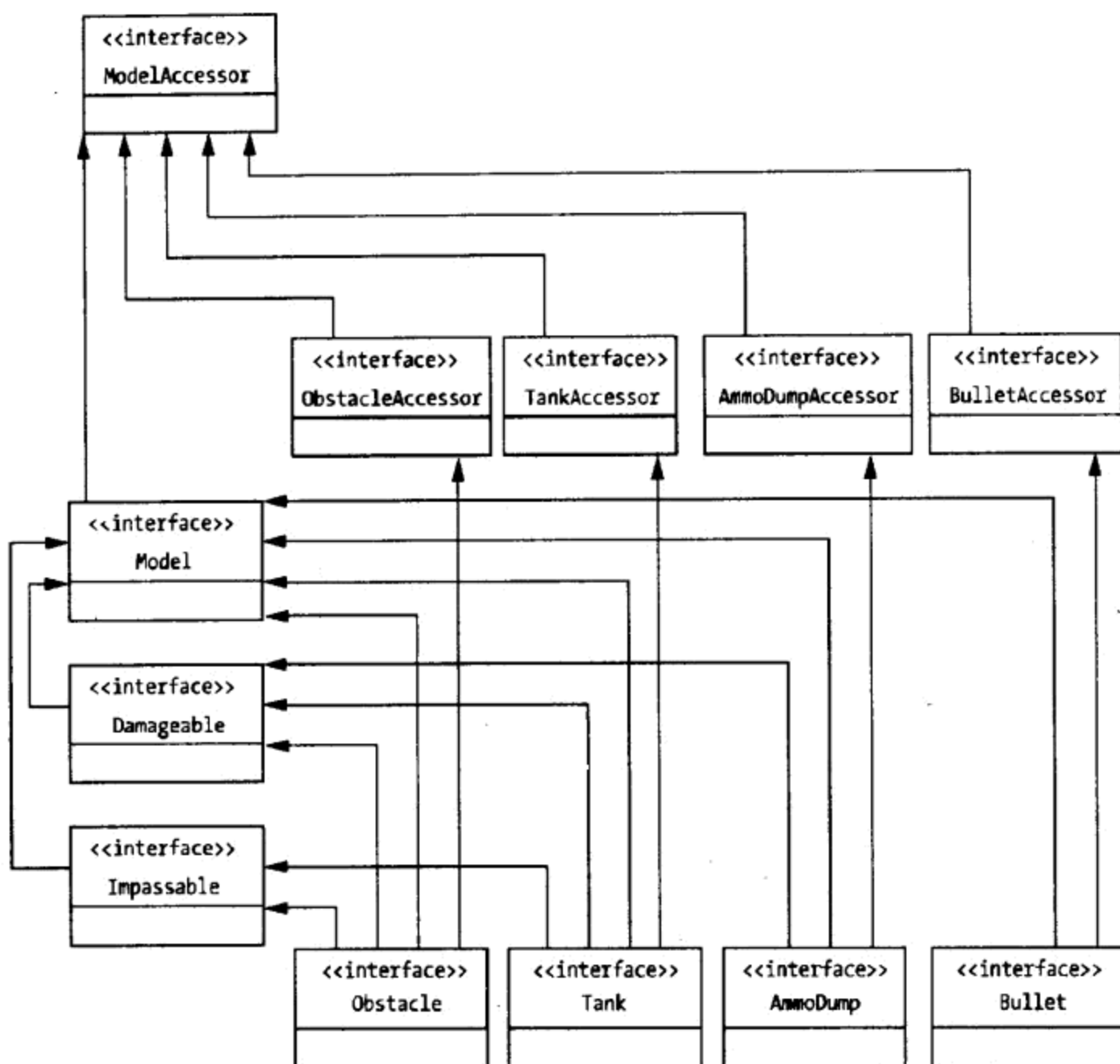


图 7-2 多重接口继承

每一个实体类型都有一个具体类的实现。因为 Model 的实例会交互，所以它们需要互相进行引用。从将来灵活性方面考虑，这些引用会限制为接口引用，而不是具体类的引用。例如 Bullet 的实现会通过特征接口 Damageable 引用 Model 的其他实现。例如，如果需要某些更加详细的实现(例如 Obstacle 或 Tank)，就会使用实体接口而不是使用具体类的接口。为了推动这种使用方式，具体类将不再提供在它们实现的接口以内可用的公共方法。

因为我们现在依赖多重接口继承，所以随着时间的推移，多重接口继承应该很容易向游戏添加其他的特征类型。这样的示例可能就是 Flammable，它表示一个实体容易受到火力的破坏。另外一个示例可能就是 Reflector，它表示子弹从一个实体上反弹。

1. SeriModel

```
package com.croftsoft.apps.mars.model.seri;

import java.io.Serializable;

import com.croftsoft.apps.mars.model.Model;

public abstract class SeriModel
```

```

    implements Comparable, Model, Serializable
    ///////////////////////////////////////////////////////////////////
    ///////////////////////////////////////////////////////////////////
    {

private static final long serialVersionUID = 0L;

    ///////////////////////////////////////////////////////////////////
    // interface Comparable method
    ///////////////////////////////////////////////////////////////////

public int compareTo ( Object other )
    ///////////////////////////////////////////////////////////////////
    {

        double otherZ = ( ( Model ) other ).getZ ( );

        double z = getZ ( );

        if ( z < otherZ )

        {
            return -1;
        }

        if ( z > otherZ )
        {
            return 1;
        }

        return 0;
    }
}

```

类 `SeriModel` 实现了接口 `Model`。该类也实现了接口 `java.lang.Comparable`，这样它就可以按照 `z` 轴次序进行存储。它实现了语义接口 `java.io.Serializable`，表示使用对象串行化的方法可以将它存储在硬盘上或通过网络发送出去。我习惯将具体类的实现分别组装在一个单独包中，该包就按照数据访问机制进行命名。在这里，该包就是 `com.croftsoft.apps.mars.model.seri`，其中 `seri` 就是串行化(`serializable`)的简写，表示使用的是简单的对象串行化机制。将来，可能会为基于其他数据访问机制(例如 `EJB`、`JDBC`、`JDO`、`JMS`、远程方法调用(`Remote Method Invocation`, `RMI`)或随机访问文件)的具体类的实现创建其他子包。但是，目前在最初开发游戏时，我假定游戏很小，小到足以在玩游戏的过程中，可以将整个对象图形都加载到内存中，并且在需要保存游戏的时候，还可以使用标准的对象串行化的方法将它写到文件中。

类 `SeriModel` 是一个抽象类，因为它会用作该包中具体类 `Model` 实现的超类。您会注意到，子类都没有继承 `protected` 类型的实例变量，并且只有一个方法——`compareTo()`——使用了在接口 `Model` 中定义的 `getZ()` 方法，以便在子类的实现中访问数据。这也反映了我使用的白盒重用是多么不可靠。这个抽象类的主要目的是为将来可能会添加到 `Model` 接口中的那些方法的实现，提供一个简单的途径。例如，假设可重用游戏库相对于现有的几百个实体类型来说有些过时。于是可以在 `Model` 接口中添加一个新的方法，例如 `getHeading()` 或 `getSpeed()`。对其中的有

些实体来说，需要修改它们具体类的实现以添加该方法。然而，对其余的那些实体，可以在它们继承的抽象超类中提供一个实现，这些抽象的超类只会返回一个默认的值或执行一个默认的操作。

2. SeriAmmoDump

```
package com.croftsoft.apps.mars.model.seri;

import java.awt.Shape;
import java.io.Serializable;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.math.geom.Circle;

import com.croftsoft.apps.mars.model.AmmoDump;
import com.croftsoft.apps.mars.model.Damageable;
import com.croftsoft.apps.mars.model.World;

public final class SeriAmmoDump
    extends SeriModel
    implements AmmoDump
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;
```

类 **SeriAmmoDump** 是游戏实体模型的具体类实现的一个很好范例。该包中的所有具体类的实现都扩展了 **SeriModel**，然后还实现一个实体特定的接口。对 **SeriAmmoDump** 而言，实体特定的接口就是 **AmmoDump**。

```
public static final class Shared
    implements Serializable
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

    //

    public static final double DEFAULT_AMMO_GROWTH_RATE = 0.5;

    public static final double DEFAULT_AMMO_MAX = 30.0;

    public static final double DEFAULT_EXPLOSION_FACTOR = 3.0;

    public static final double DEFAULT_Z = 0.1;

    //

    public final double ammoGrowthRate;
```

```

public final double ammoMax;

public final double explosionFactor;

public final double z;

////////////////////////////////////
////////////////////////////////////

public Shared (
    double ammoGrowthRate,
    double ammoMax,
    double explosionFactor,
    double z )
////////////////////////////////////
{
    this.ammoGrowthRate = ammoGrowthRate;

    this.ammoMax = ammoMax;

    this.explosionFactor = explosionFactor;

    this.z = z;
}

public Shared ( )
////////////////////////////////////
{
    this (
        DEFAULT_AMMO_GROWTH_RATE,
        DEFAULT_AMMO_MAX,
        DEFAULT_EXPLOSION_FACTOR,
        DEFAULT_Z );
}

////////////////////////////////////
////////////////////////////////////
}

```

SeriAmmoDump 定义了一个叫作 **Shared** 的静态方法。这个类包含了那些一般都被定义为游戏中的静态常量的那些变量。因为我们希望游戏的设计者能够调整这些常量而不需要修改源代码，这些常量被定义为可以从配置文件中进行初始化的非静态变量。由于大部分时间里这些变量都会保持相同的值，所以我们就创建一个 **Shared** 实例，并将其作为一个构造函数参数，在游戏中 **SeriAmmoDump** 的所有实例之间共享该实例。当然，如果希望某一个 **SeriAmmoDump** 实例的这些变量都分别有惟一值，那么 **SeriAmmoDump** 实例就需要有它自己非共享的 **Shared** 实例。

```

private final Circle circle;

private final Shared shared;

```



```

private final World world;

private final Circle explosionCircle;

//

private double ammo;

private Damageable [ ] damageables;

private boolean exploding;

private boolean updated;

```

实例变量 `circle` 中包含用于检测碰撞的 `Shape` 对象。类 `Circle` 是 `Shape` 的一个定制的实现，在下面将进一步介绍它。`World` 中包含与游戏所有 `Model` 实例的链接，它允许 `SeriAmmoDump` 实例发现和操作其他 `Model` 的实现。`explosionCircle` 是在 `AmmoDump` 被破坏的时候，爆炸所覆盖的区域。`Damageable` 数组的作用是作为一个临时存储器。变量 `ammo`、`exploding` 和 `updated` 存储在存取器接口中定义的状态。

```

public SeriAmmoDump (
    World world,
    double centerX,
    double centerY,
    double ammo,
    Shared shared )
    //////////////////////////////////////
{
    NullPointerException.check ( this.world = world );

    circle = new Circle ( );

    setCenter ( centerX, centerY );

    setAmmo ( ammo );

    NullPointerException.check ( this.shared = shared );

    damageables = new Damageable [ 0 ];

    explosionCircle = new Circle ( );
}

```

构造函数为 `final` 类型的实例变量 `world`、`circle` 和 `shared` 赋值，并将它们委托给 `mutator` 方法 `setCenter()` 和 `setAmmo()`。

```

public boolean isActive ( ) { return true; }

public Shape getShape ( ) { return circle; }

public boolean isUpdated ( ) { return updated; }

public double getZ ( ) { return shared.z; }

```

这些就是在接口 `ModelAccessor` 中定义方法的实现。方法 `isActive()` 总是会返回 `true`，原因就是 `AmmoDump` 永远不会“死掉”；它会被重置。

```
public void setCenter (
    double x,
    double y )
    //////////////////////////////////////
{
    circle.setCenter ( x, y );
}
```

这个方法和下面的那两个方法都是在接口 `Model` 中定义的。在 `Circle` 实例的情况下，由于使用了 `Shape` 方法检测碰撞和交叉点，所以在坐标空间中重新定位游戏实体也会修改该游戏实体的形状。

```
public void prepare ( )
    //////////////////////////////////////
{
    updated    = false;

    exploding = false;
}
```

方法 `prepare()` 将 `updated` 和 `exploding` 重置为 `false`。如果状态发生了改变，变量 `update` 将会被重置；而如果 `AmmoDump` 遭到了破坏，`exploding` 变量就会被重置。

```
public void update ( double timeDelta )
    //////////////////////////////////////
{
    if ( !exploding )
    {
        double newAmmo = ammo + timeDelta * shared.ammoGrowthRate;

        if ( newAmmo > shared.ammoMax )
        {
            newAmmo = shared.ammoMax;
        }
        else if ( newAmmo < 0.0 )
        {
            newAmmo = 0.0;
        }

        setAmmo ( newAmmo );
    }
}
```

调用 `update()` 方法的时候，该方法首先检查它是不是 `exploding` 的。它也可能不是 `exploding` 的，例如，如果 `Bullet` 实例在当前游戏循环迭代的更新阶段中，在 `AmmoDump` 以前就调用了它自己的 `update()` 方法，那么它就不是 `exploding` 的。如果它是 `exploding` 的，那么 `update()` 方法就什么事情也不做了，原因就是爆炸的时候，不可能向 `AmmoDump` 添加新的弹药。另一方

面弹药的数量会以某个指定的速度递增。检查 `newAmmo` 的值是不是小于 0 是为了防止有负增长或 `timeDelta` 为负数。当 `timeDelta` 为负数的时候，模拟的对象(子弹)就会向相反的方向运行，子弹就反向运行，并射中发射该子弹的坦克。

```
public void addDamage ( double damage )
////////////////////////////////////
{
    if ( exploding )
    {
        return;
    }

    updated = true;

    exploding = true;

    explosionCircle.setCenter ( circle.getCenter ( ) );

    explosionCircle.setRadius ( shared.explosionFactor * ammo );

    damageables
        = world.getDamageables ( explosionCircle, damageables );

    for ( int i = 0; i < damageables.length; i++ )
    {
        Damageable damageable = damageables [ i ];

        if ( damageable == null )
        {
            break;
        }

        damageable.addDamage ( ammo );
    }

    setAmmo ( 0.0 );
}
```

方法 `addDamage()` 是在接口 `Damageable` 中定义的。向 `AmmoDump` 中增加任意数量的破坏都会导致它们发生爆炸，除非它已经发生了爆炸。由于这是一个状态更改，所以当增加任意数量破坏的时候，布尔标志 `updated` 就会被重置。爆炸有两个方面的影响。首先，在爆炸半径以内所有具有 `Damageable` 的其他 `Model` 实例都会发生爆炸。其次，`AmmoDump` 中弹药的数量应该减少到 0。

接口 `World` 中的方法 `getDamageables()` 是用来检索游戏世界里与 `AmmoDump` 相互重叠的一组 `Damageable` 实例的。将 `explosionCircle` 以一个方法参数的形式传递进来，用来确定是否有交叉点。第二个参数 `damageables` 是一组 `Damageable`，这组 `Damageable` 会存储返回值。如果 `damageables` 太小，就会创建并返回一组新的 `Damageable`。如果它太大，给定索引位置上的一个空值将会表示有效数据的结束。这个数组是以实例变量的形式存储的，因此在下一次调用

这个方法的时候，它就可以被重新使用。这项技术减小了必须在每一个游戏循环中执行的对象创建和垃圾回收的数量。

由于调用一个实例的 `addDamage()` 方法也会调用其他实例上相同的这个 `addDamage()` 方法，所以射击一个 `AmmoDump`，可能会导致其他 `AmmoDump` 实例也发生爆炸（如果这些 `AmmoDump` 实例重叠在一起的话，就会发生连锁爆炸）。因为 `getDamageables()` 方法返回所有重叠在一起的 `Damageable` 实例（包括发起调用的 `AmmoDump` 实例），所以必须通过在 `addDamage()` 方法中的这个最初的检查来判断 `AmmoDump` 是否已经 `exploding`，来防止进行无穷递归。

```
public double getAmmo          ( ) { return ammo; }

public boolean isExploding     ( ) { return exploding; }

public Shape getExplosionShape ( ) { return explosionCircle; }
```

这些方法都是在接口 `AmmoDumpAccessor` 中定义的。方法 `getAmmo()` 只是返回 `explosionCircle`。如果将来想将爆炸的形状由圆形修改为其他某些 `Shape` 的实现——这取决于区域内起到屏障作用的建筑物，就可以直接这样修改而不需要更改接口。

```
public void setAmmo ( double ammo )
///////////////////////////////////////////////////////////////////
{
    if ( this.ammo == ammo )
    {
        return;
    }

    this.ammo = ammo;

    updated = true;

    if ( ammo > 0.0 )
    {
        circle.setRadius ( ammo );
    }
    else
    {
        circle.setRadius ( 0.0 );
    }
}
```

类 `SeriAmmoDump` 中的 `final` 方法 `setAmmo()` 是在接口 `AmmoDump` 中定义的。如果要将弹药的数量设置为与原来的值相同，这个方法就会立即返回，而不会设置 `updated` 标志，因为没有对状态进行更改。注意，`circle` 的半径以及由半径所得来的对象碰撞形状的大小，都是由 `AmmoDump` 中弹药的数量决定的。

3. Circle

```
package com.croftsoft.core.math.geom;

[...]
```



```

public class Circle
    extends [...]
    implements CircleAccessor, Serializable
    //////////////////////////////////////
    //////////////////////////////////////
    {

private static final long serialVersionUID = 0L;

//

private final Point2DD center;

//

private double radius;

////////////////////////////////////
////////////////////////////////////

public Circle (
    double centerX,
    double centerY,
    double radius )
    //////////////////////////////////////
    {
        center = new Point2DD ( centerX, centerY );

        setRadius ( radius );
    }

[...]

public boolean contains (
    double x,
    double y )
    //////////////////////////////////////
    {
        return center.distance ( x, y ) <= radius;
    }

public boolean intersectsCircle ( CircleAccessor circleAccessor )
    //////////////////////////////////////
    {
        double distance
            = center.distanceXY ( circleAccessor.getCenter ( ) );

        return distance <= radius + circleAccessor.getRadius ( );
    }

public boolean intersectsShape ( Shape shape )

```

```

////////////////////////////////////
{
    if ( shape instanceof CircleAccessor )
    {
        return intersectsCircle ( ( CircleAccessor ) shape );
    }

    if ( radius == 0.0 )
    {
        return shape.contains ( center.x, center.y );
    }

    return intersects ( shape.getBounds2D ( ) );
}

```

我喜欢使用圆形作为可以在 2D 空间中旋转实体的碰撞检测形状。正是由于这个原因，我才创建了 `java.awt.Shape` 的一个称为 `Circle` 的定制实现，该 `Circle` 的定制实现提供了专门计算交叉点的方法。可以在 `com.croftsoft.core.math.geom` 包中看到该实现的代码，这个包包含很多可重用的几何学类，例如 `PointXY` 和 `Point2DD`。

`PointXY` 是存储双精度 `x` 和 `y` 坐标的具体类的一个只读接口。`Point2DD` 是一个 `PointXY` 实现，它是内核包 `java.awt.geom` 中 `Point2D.Double` 的子类。`Point2DD` 提供了 `mutator` 方法，而 `PointXY` 没有提供这种方法。如果需要一个方法中返回一个 `Point2DD` 对象，但是不希望调用代码有机会修改该对象的话，就会定义这个方法返回接口 `PointXY` 而不是返回 `Point2DD`。

使用存取器接口防止对返回对象进行修改的另一种方法就是只返回一个副本。如果数据被复制到 `mutable` 类中，调用代码可能就会试图修改数据，但是这不会有什么影响。使用 `immutable` 具体类或者存取器接口的引用来返回数据，都会使数据不能由调用代码进行修改的这一事实变得更加明确。但是使用存取器接口引用会更加有效，因为这样既不需要将数据重新复制一次，也不需要创建一个新的对象。

类 `Circle` 使用了 `Point2DD` 的一个实例来存储中心点。`Point2DD` 提供了两个可以计算它自己和另外一个点之间距离的方法，这两个方法分别是 `distance(x,y)` 和 `distanceXY(PointXY)`。类 `Circle` 的方法 `contains(x,y)` 和 `intersectsCircle(Circle)` 使用这些距离的值确定一个点或另一个 `Circle` 的边缘是否落在它的半径以内。如果方法 `intersectsShape(Shape)` 可以确定 `Shape` 参数就是 `CircleAccessor` 的一个实例的话，它就会委托给 `intersectsCircle(CircleAccessor)` 以得到更加精确的处理。否则它就委托给方法 `intersects(Rectangle2D)`，而后者是在接口 `Shape` 中定义并从超类中继承的。

7.4.3 视图

视图对象是专门向用户呈现模型数据的。通过将模型和视图隔离的方式，我们可以很容易地将视图从一种表示形式转换为另一种表示形式。例如，我们可能会有顶视图、侧视图、等轴视图、第一人称视图或嵌套视图。我们也可能会有 2D 视图、3D 视图、低分辨率的视图、高分辨率的视图、低色彩的视图、高保真色彩的视图甚至是文本视图。

视图不必要都是可见的。只要人们能够感觉到的数据的所有表示形式都可能成为视图，这包括声音、语音、触觉、运动和嗅觉。主要思想是将视图与模型隔离开以后，可以获得很多灵

活性，这些灵活性允许在运行时切换视图或合并视图。例如可以将游戏设计为在 PDA 系统上以 2D 游戏的方式运行，而在桌面系统上以 3D 游戏的方式运行。可以在第一人称驾驶舱视图上重叠一个自顶向下的区域雷达视图。即使视图发生了改变，模型代码还是一样。

在传统的 MVC 模型中，只要模型的状态发生了改变，模型就会产生一个事件。视图将会侦听这些事件并对这些事件做出相应的反应——查询模型以获取更改后的数据。然而，在动画游戏中，模型的状态通常在每一帧中都会由主循环进行修改。因此，我们的视图将会依赖于在与主游戏循环同步中的轮询(polling)，而不是事件驱动的更新。在后面讲解多玩家联网模式一章中，将会重新介绍事件驱动的其他相关技术。

在 Mars 游戏的 2D 版中，将前面一章介绍的 ComponentAnimator 作为视图对象的一个基本接口。这意味着所有的视图类都需要实现在那个接口中定义的那两个方法 update(JComponent) 和 paint(JComponent,Graphics2D)。我们应该记得，update(JComponent)方法负责更新动画的状态，并请求 JComponent 的重绘——在需要重绘的时候。

1. ModelAnimator

```
package com.croftsoft.apps.mars.view;

import java.awt.*;
import java.awt.geom.Rectangle2D;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.lang.NullArgumentException;

import com.croftsoft.apps.mars.model.ModelAccessor;

public class ModelAnimator
    implements ComponentAnimator
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    protected final ModelAccessor modelAccessor;

    protected final Rectangle oldRepaintRectangle;

    protected final Rectangle newRepaintRectangle;

    //

    protected Color color;

    protected boolean previouslyActive;

    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    public ModelAnimator (
        ModelAccessor modelAccessor,
```

```

    Color color )
    //////////////////////////////////////
    {
        NullPointerException.check ( this.modelAccessor = modelAccessor );

        this.color = color;

        oldRepaintRectangle = new Rectangle ( );

        newRepaintRectangle = new Rectangle ( );
    }

```

类 `ModelAnimator` 可以驱动任何实现了 `Model Accessor` 接口的对象。这里之所以使用的是存取器接口 `ModelAccessor` 而不是 `Model` 接口，是由于视图不修改 `Model` 的状态。这个动画是非常常见的，因为在其基本接口 `ModelAccessor` 中返回可以显示状态数据的方法只有两个，分别是 `getShape()` 方法和 `isActive()` 方法。`shape` 用来确定屏幕上区域的边界和位置。活动标志用来确定实体是否应该被绘制或消除。

注意，在构造函数中添加了一个 `Color`。现在有许多彩色的形状运行在整个屏幕上。这确实比它发出的声音更加有用。可以在开发过程(在编写更为详细的模型特定视图类的时候)中使用 `ModelAnimator` 来测试和调试精灵的行为方式。

由于指向 `color` 的实例变量被声明为 `protected` 和非 `final` 类型的，所以该实例变量可以通过子类的实现进行更改。这是状态数据存在于视图对象中而不是存在于模型中的一个示例。即使模型相同，是否使用 `color` 也要取决于视图的实现。记住，这种分离的另一个不太关键的方面就是，一个视图的实现可能不会使用模型中所有的状态数据，即使另一个视图的实现或许可能会使用模型中所有的状态数据。一个视图可能会添加状态数据以在坦克的顶部旋转雷达的抛物面天线反射镜，而另一个视图可能会忽略模型的状态，例如坦克航向。

```

public void update ( JComponent component )
    //////////////////////////////////////
    {
        boolean currentlyActive = modelAccessor.isActive ( );

        if ( !previouslyActive )
        {
            if ( currentlyActive )
            {
                // went from off to on

                getRepaintRectangle ( oldRepaintRectangle );

                component.repaint ( oldRepaintRectangle );

                previouslyActive = currentlyActive;
            }

            // otherwise stayed off
        }
        else if ( !currentlyActive )

```



```

{
    // went from on to off

    component.repaint ( oldRepaintRectangle );

    previouslyActive = currentlyActive;
}
else
{
    // stayed on

    getRepaintRectangle ( newRepaintRectangle );

    if ( !oldRepaintRectangle.equals ( newRepaintRectangle ) )
    {
        Rectangle2D.union (
            oldRepaintRectangle,
            newRepaintRectangle,
            oldRepaintRectangle );

        component.repaint ( oldRepaintRectangle );

        oldRepaintRectangle.setBounds ( newRepaintRectangle );
    }
    else if ( isUpdated ( ) )
    {
        component.repaint ( newRepaintRectangle );
    }
}
}

protected void getRepaintRectangle ( Rectangle repaintRectangle )
///////////////////////////////////////////////////////////////////
{
    repaintRectangle.setBounds (
        modelAccessor.getShape ( ).getBounds ( ) );
}

protected boolean isUpdated ( )
///////////////////////////////////////////////////////////////////
{
    return modelAccessor.isUpdated ( );
}

public void paint (
    JComponent component,
    Graphics2D graphics )
///////////////////////////////////////////////////////////////////
{
    if ( modelAccessor.isActive ( ) )
    {
        if ( color == null )

```

```

    {
        graphics.setColor ( component.getForeground ( ) );
    }
    else
    {
        graphics.setColor ( color );
    }
    graphics.fill ( modelAccessor.getShape ( ) );
}
}

```

上面给出的 `paint()` 方法的实现很容易让人明白，但是 `update()` 方法的实现还需要进一步进行解释。首先，注意在 `update(JComponent)` 方法的内部全部都是更新的模型状态。没有调用 `Model` 实例的 `update(timeDelta)` 方法。这里的假定是，希望将模型状态的更新与视图状态的更新分隔开来。在前面几章的示例中，这两种更新都是合并在一起的，因为作为精灵来讲，它的模型和视图是合并在一起的。但是现在模型可以独立于我们的视图单独进行更新，我们也获得了更多的灵活性。例如，可以同时显示同一个模型的两个不同的视图而不需要同时两次更新模型的状态。我们也可以显示一个模型的本地视图，该本地视图的状态是在一个远程服务器的控制下进行更新的。

在处理精灵动画的时候，需要应用 `repaint()` 方法绘制当前帧内精灵相对于前一帧中该精灵的位置。当然，如果不关心帧速率性能是否令人满意，也可以不管精灵是否发生了移动，对每一帧都简单地重绘整个显示屏幕。如果这样，`update(JComponent)` 方法的实现可能就会只包含一行调用 `component.repaint()` 方法的代码。刚开始编写游戏程序的时候，这可能是最好的一种方法。如果后来发现需要其他的性能，就可以返回去再实现一些更高效的内容，例如上面给出的 `update()` 方法的示例代码。

`ModelAnimator` 的 `update()` 方法首先会检查 `ModelAccessor` 实例是否是活动的。如果 `ModelAccessor` 以前不是活动的，而现在变成活动的，那么它需要在新位置上重新绘制。如果它在前面也是活动的，而现在不再是活动的，那么就需要在原位置将它删除。如果它在前面是活动的，而现在仍然是活动的，且如果它移动了位置，那么就需要在原位置将它删除，并在新位置上将它重新绘制出来。而如果它没有移动位置，但是有一些可见的地方发生了更新，那么就需要将其重新绘制。下面这种情况就会发生这种事情：坦克体的位置没有发生变化，但是坦克炮塔发生了旋转，需要在炮塔的位置上更新动画的显示。

`update()` 方法调用了两个帮助方法 `getRepaintRectangle()` 和 `isUpdated()`，这两个方法作为单独的方法分隔出来的目的是为了它们的子类可以很容易地重写它们。`getRepaintRectangle()` 方法返回要重绘屏幕的区域。默认的行为是返回整个碰撞区域的一个矩形。如果视图区域是在基于 `Model` 的一个不同的比例上，或完全不同于它虚拟的物理形式，那么将需要重写这个方法。方法 `isUpdated()` 确定精灵是否应该进行重绘——即使该精灵没有移动。该默认的行为是假定当 `Model` 的状态被更新的时候，视图也应该进行更新。如果不是这种情况，尤其是如果有一些没有表现出来的更新状态的时候，就要重写这个方法。

`update()` 方法调用 `Rectangle2D.union()` 将原先的重绘区域与新重绘区域合并在一起。这里假设重绘区域由于假定运动是连续的和递增的而相互叠加。如果不是这种情况，在优化上的努力就会导致性能的剧烈下降。这种情况是可能发生的，例如，如果有一个这样的游戏实体：该实体可以在屏幕上进行超时空转移。在这种情况下，就要分别重绘该实体原位置和新位置。

2. AmmoDumpAnimator

```

package com.croftsoft.apps.mars.view;

import java.awt.*;
import javax.swing.JComponent;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.media.sound.AudioClipCache;

import com.croftsoft.apps.mars.model.AmmoDumpAccessor;

public final class AmmoDumpAnimator
    extends ModelAnimator
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final Color COLOR                                = Color.YELLOW;

    private static final Color HIT_COLOR                            = Color.RED;

    private static final String EXPLOSION_AUDIO_FILENAME = "explode.wav";

    //

    private final AmmoDumpAccessor ammoDumpAccessor;

    private final AudioClipCache audioClipCache;

    //////////////////////////////////////
    //////////////////////////////////////

    public AmmoDumpAnimator (
        AmmoDumpAccessor ammoDumpAccessor,
        AudioClipCache audioClipCache )
    //////////////////////////////////////
    {
        super ( ammoDumpAccessor, COLOR );

        this.ammoDumpAccessor = ammoDumpAccessor;

        NullArgumentException.check (

            this.audioClipCache = audioClipCache );
    }

    //////////////////////////////////////
    //////////////////////////////////////

    public void update ( JComponent component )
    //////////////////////////////////////

```

```

{
    super.update ( component );

    if ( ammoDumpAccessor.isExploding ( ) )
    {
        audioClipCache.play ( EXPLOSION_AUDIO_FILENAME );
    }
}

protected void getRepaintRectangle ( Rectangle repaintRectangle )
///////////////////////////////////////////////////
{
    if ( ammoDumpAccessor.isExploding ( ) )
    {
        repaintRectangle.setBounds (
            ammoDumpAccessor.getExplosionShape ( ).getBounds ( ) );
    }
    else
    {
        super.getRepaintRectangle ( repaintRectangle );
    }
}

public void paint (
    JComponent component,
    Graphics2D graphics )
///////////////////////////////////////////////////
{
    if ( ammoDumpAccessor.isExploding ( ) )
    {
        graphics.setColor ( HIT_COLOR );

        graphics.fill ( ammoDumpAccessor.getExplosionShape ( ) );
    }
    else
    {
        super.paint ( component, graphics );
    }
}

```

AmmoDumpAnimator 通过添加特殊的代码来扩展 **ModelAnimator**，以使一个 **AmmoDumpAccessor** 发生爆炸。由于只会调用 `update()` 一次而会多次调用 `paint()` 方法，所以这个爆炸的声音是在 `update()` 方法的内部播放的。例如，如果爆炸的声音是在 `paint()` 方法中播放，而游戏又暂停的时候，那么由于窗口事件而导致的每次重绘就都会听到爆炸的声音。

7.4.4 控制器

控制器对象捕获用户的输入并将它们作为操纵状态的请求一直传递到模型里。通过将控制器与模型分离，我们将模型与用户使用的输入设备独立开来。输入设备可以包含键盘、鼠标、话筒和网络。


```

public void keyPressed ( KeyEvent keyEvent )
////////////////////////////////////
{
    if ( keyEvent.getKeyCode ( ) == KeyEvent.VK_SPACE )
    {
        if ( ammo > 0 )
        {
            ammo--;

            firing = true;
        }
        else
        {
            dryFiring = true;
        }
    }
}

```

作为示例，类可以直接侦听键盘事件，只要按下空格键，坦克精灵的状态就会变成开火射击。用户实际上是否能够开火要视游戏模型准许的内容而定。在这个示例中，模型的逻辑会检查是否有足够的弹药进行开火。

```

public void keyPressed ( KeyEvent keyEvent )
////////////////////////////////////
{
    if ( keyEvent.getKeyCode ( ) == KeyEvent.VK_SPACE )
    {
        requestFire ( );
    }
}

public void mousePressed ( MouseEvent mouseEvent )
////////////////////////////////////
{
    requestFire ( );
}

private void requestFire ( )
////////////////////////////////////
{
    if ( ammo > 0 )
    {
        ammo--;

        firing = true;
    }
    else
    {
        dryFiring = true;
    }
}

```

如果想添加也能使用鼠标进行开火的选项，就要创建一个间接层来防止代码的复制——将请求处理逻辑分隔到一个单独的方法中。在这个示例中，私有方法 `requestFire()` 现在就包含了处理状态更新请求的模型逻辑。

现在进一步假定想给玩家一个可以自己定义键盘控制的选项，这样用户就可以使用 `Ctrl` 键代替空格键进行开火。这就会需要另一个可能会提高类复杂性的间接层。如果有某些用户的输入可能会来自于网络的这种可能性，那么干脆考虑将这个代码移动到一个或多个单独的控制器对象中。

```
package com.croftsoft.apps.mars.controller;

import java.awt.*;
import java.awt.event.*;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.gui.event.UserInputAdapter;
import com.croftsoft.core.math.geom.Point2DD;

import com.croftsoft.apps.mars.ai.TankOperator;

public final class TankController
    extends UserInputAdapter
    //////////////////////////////////////
    //////////////////////////////////////
{

    private final TankOperator tankOperator;

    private final Point2DD destination;

    //////////////////////////////////////
    //////////////////////////////////////

    public TankController (
        TankOperator tankOperator,
        Component component )
    //////////////////////////////////////
    {
        NullArgumentException.check ( this.tankOperator = tankOperator );

        NullArgumentException.check ( component );

        component.addKeyListener ( this );

        component.addMouseListener ( this );

        component.addMouseMotionListener ( this );

        component.requestFocus ( );

        destination = new Point2DD ( );
    }
}
```



```

////////////////////////////////////
////////////////////////////////////

public void keyPressed ( KeyEvent keyEvent )
////////////////////////////////////
{
    tankOperator.fire ( );
}

public void mouseMoved ( MouseEvent mouseEvent )
////////////////////////////////////
{
    Point mousePoint = mouseEvent.getPoint ( );

    destination.setXY ( mousePoint.x, mousePoint.y );

    tankOperator.go ( destination );
}

public void mousePressed ( MouseEvent mouseEvent )
////////////////////////////////////
{
    tankOperator.fire ( );
}

```

TankController 扩展了 com.croftsoft.core.gui.event 包中的 UserInputAdapter。类 UserInputAdapter 是一个实现内核接口 KeyListener 和 MouseInputListener 的适配器类。只要按下了键盘或单击了鼠标，就会调用 TankOperator 实例的 fire()方法。在后面有关人工智能的一章中还要进一步介绍 TankOperator，但是现在可以假定 TankOperator 是 Tank 模型对象的一个代理。这个示例的主要点就是演示了输入设备的代码从它控制的 Model 类中分离出去。

7.5 复合 MVC

以上的那些示例解释了怎样可以将游戏世界里的单个实体(例如军火库或坦克)分解到模型、视图和控制类中。下面的示例将演示应该怎样使用这些单独的实体组件，构成游戏世界的复合 MVC。

7.5.1 复合模型

简单地说，复合模型就是由其他模型组合而成的一个模型。如果有一个组件-子组件的层次结构，而在这个层次结构中，子组件如果脱离了整个层次就不存在，复合模型就显得非常有用。例如，可以按照坦克体、坦克炮塔、履带和弹药，将坦克模型分解到几个单独的类中。然而，我并没有进行这种分解，因为好像没有必要这么做。或许以后我发现可以重用其中的某一个单独的子组件时，我才会那样分解。我没有将坦克子弹的模型作为游戏世界中一个独立的部分分解出来，是因为一旦开火，它就可以独立于坦克而独立存在。对 Mars 游戏而言，主要的复合模型的示例就是 World 和 Game。

1. WorldAccessor

```
package com.croftsoft.apps.mars.model;

public interface WorldAccessor
////////////////////////////////////
////////////////////////////////////
{

    public ModelAccessor [ ] getModelAccessors (
        ModelAccessor [ ] modelAccessors );
```

World Accessor接口提供了一个方法来获取所有Model实例(无论是否是活动的, 只要是World中的实例)的存取器接口。它主要由视图使用。

2. World

```
package com.croftsoft.apps.mars.model;

[...]
```

```
public interface World
    extends WorldAccessor
////////////////////////////////////
////////////////////////////////////
{
    public void clear ( );

    public AmmoDump createAmmoDump (
        double centerX,
        double centerY );

    [...]

    //////////////////////////////////////
    //////////////////////////////////////

    public void prepare ( );

    public void update ( double timeDelta );

    //////////////////////////////////////
    //////////////////////////////////////

    public void fireBullet (
        double originX,
        double originY,
        double heading );

    [...]

    public Damageable [ ] getDamageables (
```



```

    Shape shape,
    Damageable [ ] damageables );

[...]
```

```
public boolean isBlocked ( Model model );
```

接口 **World** 扩展了 **WorldAccessor**。这是要传递到各个具体 **Model** 实现中，以使它们可以发现并操纵其他游戏实体的那个接口。例如 **getDamageables()** 方法就是由 **SeriAmmoDump** 用来发现爆炸半径以内的其他 **Damageable** 实体的。**fireBullet()** 方法将会激活或创建一个 **Bullet** 实例，并将它添加到 **World** 中。**isBlocked()** 方法决定 **Model** 是否正在试图移动到一个已经被 **Impassable** 实体占据的位置。

上面并没有给出接口中其他一些方法。一般的原则是将那些需要激活、创建或捕获游戏中其他 **Model** 实例的方法以及发起调用的方法都放在 **World** 接口中。这些方法是怎样由 **World** 接口的一个具体类实现的，要高度依赖于其底层的数据访问机制。例如，一个实现可能会使用对象查询语言(Object Query Language, OQL)仅将那些 **Model** 实例存储到内存中。另一个实现可能会通过就近原则去聚集 **Model** 实例。所有这些优化处理的技术都应该隐藏在接口背后，这样才可以在不破坏调用代码的情况下调整它们。

3. SeriWorld

```

package com.croftsoft.apps.mars.model.seri;

[...]
```

```

public final class SeriWorld
    implements World, Serializable
    //////////////////////////////////////
    //////////////////////////////////////
    {

private static final long serialVersionUID = 0L;

[...]
```

```

private Model [ ] getModels ( )
    //////////////////////////////////////
    {
        [...]
    }

[...]
```

```

public void prepare ( )
    //////////////////////////////////////
    {
        Model [ ] models = getModels ( );

        for ( int i = 0; i < models.length; i++ )
```

```

        {
            models [ i ].prepare ( );
        }
    }
    public void update ( double timeDelta )
    //////////////////////////////////////
    {
        Model [ ] models = getModels ( );

        for ( int i = 0; i < models.length; i++ )
        {
            models [ i ].update ( timeDelta );
        }
    }

    [...]

```

我不准备深入研究 World 接口的这个具体实现——SeriWorld，但是会向您介绍您可能在几乎所有复合模型的实现中都使用的两个方法。`prepare()`和 `update(timeDelta)`方法委托给 World 中包含的所有 Model 实例。在游戏循环的更新阶段中，在调用任何一个 `update(timeDelta)`方法以前，首先对游戏中每一个单独的 Model 实例调用 `prepare()`方法是非常重要的。如果没有这样去做，那么最终模型的正常更新还依赖于它们执行的顺序。

例如，假设在游戏循环的一个迭代过程中，首先准备和更新 Bullet，再准备和更新 AmmoDump。调用 Bullet 的 `update()`方法的时候，它也调用了 AmmoDump 的 `addDamage()`方法。AmmoDump 的 `addDamage()`方法设置 `updated` 和 `exploding` 标志。在 Bullet 的 `update()`方法返回的时候，再调用 AmmoDump 的 `prepare ()`方法。在游戏循环的更新阶段结束以后，且在视图检测到爆炸和显示出爆炸以前，`prepare ()`方法会重新设置 `updated` 和 `exploding` 标志。而这通常是不符合要求的。

可以通过在 Bullet 以前首先就准备和更新 AmmoDump 防止这个问题。这能很好地解决这个问题，因为在视图有机会检测和显示爆炸以前，`exploding` 标志是不会被重新设置的。如果硬要决定采取这种方法，或许应该也将 `prepare()`方法和 `update()`合并为一个单独的方法，例如叫 `prepareAndUpdate()`，因为在更新以前，就不再需要首先将所有 Model 实例都准备在一起。但是，我认为您会发现，由于游戏中模型类型和模型交互的递增，决定执行顺序的因素也会相应增加。最终可能会得到一个让人很迷惑的环境，或者必须使用许多需要持续到游戏循环的下一个迭代中的专门的状态转换标志。通过开始就将准备和更新两个步骤分开，就完全可以避免这种复杂性，即使刚开始可能并不需要这样做也是如此。

4. GameAccessor

```

package com.croftsoft.apps.mars.model;

import java.util.*;

public interface GameAccessor
    //////////////////////////////////////
    //////////////////////////////////////
    {

```



```

public int          getLevel          ( );

public Iterator     getPath           ( );

public TankAccessor getPlayerTankAccessor ( );

public WorldAccessor getWorldAccessor ( );

```

Game 中包含 World。我喜欢将 Game 作为一个超自然的实体进行考虑。一旦创建了 World，它就会按照由 Model 的交互作用决定的它自身的内部规律，很好地运转。但是，需要首先用 Game 创建一个 World，并将它填充上 Model 实例。Game 也会监控 World 的状态以确定什么时候到达了终止时间，例如已经完成了一关和必须创建一个新 World 的时候。在此以前，World 的所有操作我认为都是只能听天由命，因为它存在于模型和控制器的正常交互以外。

接口 GameAccessor 提供了访问 World 数据以及其他 World 内没有以 Model 的形式存储的附加信息，例如当前游戏关信息以及玩家的一些信息。例如，对 World 而言，属于某个玩家的 Tank 看起来与所有 Tank 都一样。但是，Game 必须为控制器和专门的视图维护这种惟一标识。

5. Game

```

package com.croftsoft.apps.mars.model;

import com.croftsoft.core.animation.clock.Timekeeper;

import com.croftsoft.apps.mars.ai.TankOperator;

public interface Game
    extends GameAccessor
    //////////////////////////////////////
    //////////////////////////////////////
{

    public Tank          getPlayerTank          ( );

    public double        getTimeFactorDefault   ( );

    public Timekeeper    getTimekeeper          ( );

    //

    public void update ( );

```

接口 Game 扩展了 GameAccessor。它提供了其他一些视图并不需要的 accessor 方法，其中的某些方法返回可变的对象引用。它还包含主要的 update() 方法。

6. SeriGame

```

package com.croftsoft.apps.mars.model.seri;

[...]
```

```

import com.croftsoft.core.animation.clock.HiResClock;
import com.croftsoft.core.animation.clock.Timekeeper;
import com.croftsoft.core.lang.NullArgumentException;

[...]

public final class SeriGame
    implements Game, Serializable
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

```

类 **SeriGame** 是接口 **Game** 的一个 **Serializable** 实现。

```

public static final double DEFAULT_TIME_FACTOR          = 1.0;

public static final long  DEFAULT_RANDOM_SEED          = 1968L;

[...]

public static final double DEFAULT_OBSTACLE_RADIUS_MIN = 10.0;

```

该游戏的常量很多。这里我只给出其中的 3 个。

```

private final double timeFactorDefault;

private final long    randomSeed;

[...]

private final double obstacleRadiusMin;

```

每一个默认的常量都有一个相应的变量。通过使用变量代替常量的方法，我们就可以在初始化过程中从配置文件中加载值。

```

private final Random actionsRandom;

private final Random contentRandom;

```

实例变量 **actionsRandom** 用来为游戏中实体的行为产生随机数字，例如敌人的坦克是否应该在一个给定的时刻开火。变量 **contentRandom** 用来产生游戏关的内容，例如障碍物的位置和弹药的存放处等。我为此专门使用了一个单独的随机数生成器，因为我希望每次玩该游戏的时候，每一关的内容都相同。变量 **actionsRandom** 不能用于这个目的，因为到第二关的时候，它在随机数产生序列里的位置就会根据玩家的行为而发生变化。

```

private final TankOperator playerTankOperator;

private final Timekeeper    timekeeper;

```



```

private final World      world;

//

private int level;

private Tank playerTank;

```

指向 `playerTankOperator` 的引用在游戏关之间维持，因为 `TankController` 与它链接在一起。`Timekeeper` 用来计算和存储传递给 `update(timeDelta)` 方法的 `timeDelta`。使用 `World` 的一个接口引用而不是指向 `SeriWorld` 的一个具体类的引用是为了获得更大的灵活性。维持了 `playerTank` 引用，这样我们就可以在复合视图中将其与 `World` 里的其他 `Tank` 区分开来。

```

public static void main ( String [ ] args )
///////////////////////////////////////////////////////////////////
{
    SeriGame seriGame = new SeriGame ( );

    int level = 0;

    while ( true )
    {
        if ( seriGame.getLevel ( ) != level )
        {
            level = seriGame.getLevel ( );

            System.out.println ( new Date ( ) + ": Level " + level );
        }

        seriGame.update ( );
    }
}

```

表明已经做了一件很完美工作(将控制器和视图从模型中分隔出来并删除了所有相关性)的一个标志就是，可以不使用控制器和视图运行这个游戏。这就可以高速模拟游戏并记录游戏结果(为了对游戏进行分析)。在这里，将会打开 `PlayerTankOperator` 的自动驾驶模式，并在没有玩家输入控制的时候，使用人工智能来玩这个游戏。游戏到达下一关的时候，系统输出“视图”将会打印出日期和时间。

```

public SeriGame (
    double      timeFactorDefault,
    long        randomSeed,
    [...]
    double      obstacleRadiusMin,
    SeriAmmoDump.Shared seriAmmoDumpShared )
///////////////////////////////////////////////////////////////////
{
    this.timeFactorDefault = timeFactorDefault;

    this.randomSeed      = randomSeed;
}

```

```

[... ]

this.obstacleRadiusMin = obstacleRadiusMin;

timekeeper
    = new Timekeeper ( new HiResClock ( ), timeFactorDefault );

contentRandom = new Random ( randomSeed );

actionsRandom = new Random ( );

world = new SeriWorld (
    actionsRandom,
    seriAmmoDumpShared );

DefaultTankOperator defaultTankOperator
    = new DefaultTankOperator ( actionsRandom );

playerTankOperator = new PlayerTankOperator ( defaultTankOperator );

level = 1;

createLevel ( );
}

```

在这个例子中，游戏的参数是以构造函数参数的方式传入的。更好的方法是可以从 XML 文件读入的初始化器对象的方式将它们传入。这里使用 `timeFactorDefault` 控制整体的模拟速度，同时也可以调整 `timeFactorDefault`。如果游戏时间流逝的速度与真实时间流逝的速度不同，这就非常有用。当玩家接近一个新关的时候，它也可以用来增加一个动作游戏的困难度。变量 `actionsRandom`、`seriAmmoDumpShared` 和 `tankOperator` 也是以构造函数参数的形式传递给 `SeriWorld` 的，这样 `SeriWorld` 就可以使用它们利用它的工厂方法(例如 `createAmmoDump()` 和 `createTank()`)创建新的 Model 实例。

```

public SeriGame ( )
///////////////////////////////////////////////////
{
    this (
        DEFAULT_TIME_FACTOR,
        DEFAULT_RANDOM_SEED,
        [...]
        DEFAULT_OBSTACLE_RADIUS_MIN,
        new SeriAmmoDump.Shared ( ) );
}

```

不带参数的构造函数为主构造函数提供了默认的值。

```

public Iterator    getPath          ( ) {
    return playerTankOperator.getPath ( ); }

public int         getLevel         ( ) { return level; }

```



```

public TankAccessor getPlayerTankAccessor ( ) { return playerTank; }

public Tank      getPlayerTank ( ) { return playerTank; }

public Timekeeper getTimekeeper ( ) { return timekeeper; }

public WorldAccessor getWorldAccessor ( ) { return world; }

public double getTimeFactorDefault ( ) { return timeFactorDefault; }

```

这些都是在接口GameAccessor和Game中定义的accessor方法。

```

public void update ( )
///////////////////////////////////////////////////////////////////
{
    world.prepare ( );

    timekeeper.update ( );

    double timeDelta = timekeeper.getTimeDelta ( );

    if ( timeDelta > timeDeltaMax )
    {
        timeDelta = timeDeltaMax;
    }

    world.update ( timeDelta );
}

```

update()方法首先准备并更新 World。timeDelta 被限制在 timeDeltaMax 以内，这样游戏循环中的任何暂停或减慢速度都不会致使游戏的增量过大。例如，如果坦克的旋转速度是 1 弧度每秒，而 timeDeltaMax 是 0.2 秒，那么坦克在一个单独的帧中可以产生的最大旋转将会是 0.2 弧度。

```

Tank [ ] tanks = world.getTanks ( );

for ( int i = 0; i < tanks.length; i++ )
{
    Tank tank = tanks [ i ];

    if ( !tank.isActive ( ) )
    {
        for ( int j = 0; j < attemptsMax; j++ )
        {
            tank.initialize (
                worldWidth * actionsRandom.nextDouble ( ),
                worldHeight * actionsRandom.nextDouble ( ) );

            if ( !world.isBlocked ( tank ) )
            {
                break;
            }
        }
    }
}

```

```

    }
    }
}

```

如果 World 中一个 Tank 被摧毁了，那么 Game 就会使用它的超能力来修复它，并将它放在一个新的随机位置上。Game 将会接近 attemptsMax，以尝试将 Tank 移动到还没有被占领的一个位置上。

```

    Obstacle [ ] obstacles = world.getObstacles ( );

    for ( int i = 0; i < obstacles.length; i++ )
    {
        if ( obstacles [ i ].isActive ( ) )
        {
            return;
        }
    }
    level++;

    createLevel ( );
}

```

在它的 update()方法中，Game 也会监控表示某一关已经完成的条件。在这个游戏中，在达到下一关以前，必须摧毁所有的障碍物。

```

private void createLevel ( )
///////////////////////////////////////////////////
{
    actionsRandom.setSeed ( level );

    world.clear ( );

    playerTank = world.createTank (
        initialPlayerX,
        initialPlayerY,
        friendColor );

    playerTankOperator.setTankConsole ( playerTank );

    playerTank.setTankOperator ( playerTankOperator );
}

```

私有方法 createLevel()首先设置随机种子，这样 NPC 实体的起始行为对一个特定的关而言将始终相同。经过最初的几秒以后，NPC 实体做的事情再取决于玩家的行为。一个 World 被清除后，要为游戏关添加新的 Model 实例。为玩家添加一个新的 Tank，再分配一个 playerTankOperator 来控制它。记住，TankController 有对 playerTankOperator 的一个永久引用。

```

Rectangle driftBounds
    = new Rectangle ( 0, 0, worldWidth, worldHeight );

for ( int i = 0; i < obstacles; i++ )

```



```

{
    double radius = ( obstacleRadiusMax - obstacleRadiusMin )
        * contentRandom.nextDouble ( ) + obstacleRadiusMin;

    Obstacle obstacle = world.createObstacle (
        worldWidth * contentRandom.nextDouble ( ),
        worldHeight * contentRandom.nextDouble ( ),
        radius,
        obstacleRadiusMin,
        driftBounds );

    for ( int j = 0; j < attemptsMax; j++ )
    {
        if ( !world.isBlocked ( obstacle ) )
        {
            break;
        }

        obstacle.setCenter (
            worldWidth * contentRandom.nextDouble ( ),
            worldHeight * contentRandom.nextDouble ( ) );
    }
}

```

障碍物实例被添加到 World 中去。随着时间的流逝，它们会慢慢地在由 `driftBounds` 限制的一个区域中四处漂移。

```

for ( int i = 0; i < ammoDumps; i++ )
{
    [...]
}

```

`AmmoDump` 实例也是以类似的方式添加到 World 中去的。

```

for ( int i = 0; i < level; i++ )
{
    Tank tank = world.createTank (
        ( i + 1 ) * worldWidth,
        ( i + 1 ) * worldHeight,
        enemyColor );
}

```

敌人坦克的数量取决于 `Game` 的当前 `level`。刚开始的时候，将它们从动作中隔离开，这样使得每一个其他的坦克到达这个场景都需要花一些时间。

```

for ( int i = 0; i < level - 1; i++ )
{
    Tank tank = world.createTank (
        -( i + 10 ) * worldWidth,
        -( i + 10 ) * worldHeight,
        friendColor );
}
}

```

如果等待时间足够长，友军就会来帮助您。注意，我这里的不足之处就是使用了一个字面值 10，而不是一个可以自定义的常量或变量。

7.5.2 复合视图

复合视图就是由其他视图组成的一个视图。这个示例游戏使用了两个复合视图，即 WorldAnimator 和 GameAnimator。

1. WorldAnimator

```
package com.croftsoft.apps.mars.view;
import java.awt.*;
import java.util.*;
import javax.swing.JComponent;

import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.animation.painter.ColorPainter;
import com.croftsoft.core.awt.image.ImageCache;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.media.sound.AudioClipCache;

import com.croftsoft.apps.mars.model.AmmoDumpAccessor;
import com.croftsoft.apps.mars.model.BulletAccessor;
import com.croftsoft.apps.mars.model.ModelAccessor;
import com.croftsoft.apps.mars.model.ObstacleAccessor;
import com.croftsoft.apps.mars.model.TankAccessor;
import com.croftsoft.apps.mars.model.WorldAccessor;

public final class WorldAnimator
    implements ComponentAnimator
{
    ////////////////////////////////////////
    ////////////////////////////////////////

    private static final Color COLOR_BULLET = Color.MAGENTA;
    private static final Color COLOR_OBSTACLE = Color.BLACK;

    //

    private final AudioClipCache audioClipCache;

    private final ImageCache imageCache;

    private final Map componentAnimatorMap;

    private final WorldAccessor worldAccessor;

    //

    private ModelAccessor [ ] modelAccessors;

    ////////////////////////////////////////
    ////////////////////////////////////////
```



```

public WorldAnimator (
    WorldAccessor worldAccessor,
    AudioClipCache audioClipCache,
    ImageCache imageCache )
    //////////////////////////////////////
{
    NullPointerException.check ( this.worldAccessor = worldAccessor );

    NullPointerException.check ( this.audioClipCache = audioClipCache);

    NullPointerException.check ( this.imageCache = imageCache );

    componentAnimatorMap = new HashMap ( );

    modelAccessors = new ModelAccessor [ 0 ];
}

```

WorldAnimator 是一个 **ComponentAnimator** 接口的实现，这个实现驱动 **World** 中所有的 **Model** 实例。它将适当类型的一个 **ComponentAnimator** 视图映射到每一个 **Model**，并在 **componentAnimatorMap** 中存储这种关系以备将来使用。为了简单起见，它不从 **HashMap** 中移除 **ComponentAnimator**，即使在游戏中不会再使用这个 **Model**。下面这个事实把这个问题变得更加复杂：变为不活动的一个 **Model** 必须至少还要被视图驱动一次，以便于在原来位置上的精灵从内存中删除以前，将其在视图中删除。

```

public void update ( JComponent component )
    //////////////////////////////////////
{
    modelAccessors = worldAccessor.getModelAccessors ( modelAccessors );

    for ( int i = 0; i < modelAccessors.length; i++ )
    {
        ModelAccessor modelAccessor = modelAccessors [ i ];

        if ( modelAccessor == null )
        {
            break;
        }

        ComponentAnimator componentAnimator
            = getComponentAnimator ( modelAccessor );

        componentAnimator.update ( component );
    }
}

```

方法 **update(JComponent)** 在一个从 **worldAccessor** 中检索到的 **ModelAccessor** 数组上进行循环迭代。方法 **getComponentAnimator()** 用来为每一个 **ModelAccessor** 获取或创建 **ComponentAnimator**。然后再将 **update(JComponent)** 的调用委托。

```

public void paint (
    JComponent component,
    Graphics2D graphics )
    //////////////////////////////////////
{
    for ( int i = 0; i < modelAccessors.length; i++ )
    {
        ModelAccessor modelAccessor = modelAccessors [ i ];

        if ( modelAccessor == null )
        {
            break;
        }

        ComponentAnimator componentAnimator
            = getComponentAnimator ( modelAccessors [ i ] );
        componentAnimator.paint ( component, graphics );
    }
}

```

方法 `paint()` 几乎相同，只是它重用了在调用 `update()` 的过程中从 `WorldAccessor` 中取到的 `ModelAccessor` 数组。`ModelAccessor` 数组是为进行绘制而已经按 `z` 轴次序排序的。这种排序是由 `World` 执行和维护的，并且仅在首次创建新的 `Model` 和将它添加到 `World` 中去的时候才可能发生。

```

private ComponentAnimator getComponentAnimator (
    ModelAccessor modelAccessor )
    //////////////////////////////////////
{
    ComponentAnimator componentAnimator = ( ComponentAnimator )
        componentAnimatorMap.get ( modelAccessor );

    if ( componentAnimator == null )
    {
        if ( modelAccessor instanceof AmmoDumpAccessor )
        {
            componentAnimator = new AmmoDumpAnimator (
                ( AmmoDumpAccessor ) modelAccessor, audioClipCache );
        }
        else if ( modelAccessor instanceof BulletAccessor )
        {
            componentAnimator
                = new ModelAnimator ( modelAccessor, COLOR_BULLET );
        }
        else if ( modelAccessor instanceof ObstacleAccessor )
        {
            componentAnimator
                = new ModelAnimator ( modelAccessor, COLOR_OBSTACLE );
        }
        else if ( modelAccessor instanceof TankAccessor )
        {
            componentAnimator = new TankAnimator (
                ( TankAccessor ) modelAccessor, imageCache, audioClipCache );
        }
    }
}

```



```

    }
    else
    {
        componentAnimator = new ModelAnimator ( modelAccessor );
    }
    componentAnimatorMap.put ( modelAccessor, componentAnimator );
}

return componentAnimator;
}

```

这个 `private()` 方法将 `ModelAccessor` 参数作为键，从 `componentAnimatorMap` 中为模型检索相应的视图。当第一次处理一个具体的 `ModelAccessor` 实例的时候，`private()` 方法将会为它创建一个视图，并将它存储在 `Map` 中。只要可用，就会使用一个实体特定的实现，例如 `AmmoDumpAnimator` 或 `TankAnimator`。否则 `private()` 方法就会为未知的或简单的实体类型(例如 `BulletAccessor` 和 `ObstacleAccessor`)使用 `ModelAnimator`。

2. GameAnimator

```

package com.croftsoft.apps.mars.view;

[...]

import com.croftsoft.core.animation.ComponentAnimator;
import com.croftsoft.core.animation.painter.ColorPainter;
import com.croftsoft.core.awt.image.ImageCache;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.media.sound.AudioClipCache;

[...]

public final class GameAnimator
    implements ComponentAnimator
    //////////////////////////////////////
    //////////////////////////////////////
    {

    private final GameAccessor    gameAccessor;

    private final AudioClipCache  audioClipCache;

    private final ColorPainter    backgroundColorPainter;

    private final ImageCache      imageCache;

    //

    private int                    oldLevel;

    private PathAnimator          pathAnimator;

```

```
private TankAmmoAnimator    tankAmmoAnimator;

private WorldAnimator       worldAnimator;
```

GameAnimator 是一个接口 ComponentAnimator 的实现，它用来表示 GameAccessor 的一个视图。它创建一个 AudioClipCache 和一个 ImageCache，而且和它的委托视图 PathAnimator、TankAmmoAnimator 和 WorldAnimator 共享。PathAnimator 显示玩家坦克计划行驶的路径。TankAmmoAnimator 在屏幕一角显示玩家 Tank 的弹药指示器。一般说来，WorldAnimator 应该只驱动 World 中的 Model 实例。GameAnimator 负责将 WorldAnimator 和用户接口其余部分联合在一起。

```
public GameAnimator (
    GameAccessor gameAccessor,
    JComponent   component,
    ClassLoader  classLoader,
    String       mediaDir,
    Color        backgroundColor )
////////////////////////////////////
{
    NullPointerException.check ( this.gameAccessor = gameAccessor );

    NullPointerException.check ( component );

    audioClipCache = new AudioClipCache ( classLoader, mediaDir );

    imageCache = new ImageCache (
        Transparency.BITMASK,
        component,
        classLoader,
        mediaDir );

    backgroundColorPainter = new ColorPainter ( backgroundColor );

    updateAnimators ( component );
}

[...]
```

构造函数最后调用 updateAnimators()方法，该方法将会使用 GameAccessor 中的数据，创建最初的 WorldAnimator 和 TankAmmoAnimator 实例。

```
public void update ( JComponent component )
////////////////////////////////////
{
    int level = gameAccessor.getLevel ( );

    if ( level != oldLevel )
    {
        oldLevel = level;

        updateAnimators ( component );
    }
}
```



```

        component.repaint ( );
    }

    worldAnimator.update ( component );

    if ( pathAnimator != null )
    {
        pathAnimator.update ( component );
    }

    if ( tankAmmoAnimator != null )
    {
        tankAmmoAnimator.update ( component );
    }
}

```

`update(JComponent)`方法会监控游戏的 level, 以便检查它是否发生了改变。如果它发生了改变, `update(JComponent)`方法就会调用 `updateAnimators()`方法, 准备显示新关的数据。然后它还会请求一个全屏的刷新。然而在大部分时间里, `update(JComponent)`都只会简单地将对 `update(JComponent)`的调用委托给 `worldAnimator`、`pathAnimator` 和 `tankAmmoAnimator`。如果 Game 运行在没有玩家 Tank 的模式下, `pathAnimator` 和 `tankAmmoAnimator` 实例可能就为空。如果游戏运行在演示模式下, 或玩家 Tank 实例在销毁以后已从 World 中删除以后, 就会发生这种情况。

```

public void paint (
    JComponent component,
    Graphics2D graphics )

    //////////////////////////////////////
{
    backgroundColorPainter.paint ( component, graphics );

    worldAnimator          .paint ( component, graphics );

    if ( pathAnimator != null )
    {
        pathAnimator          .paint ( component, graphics );
    }

    if ( tankAmmoAnimator != null )
    {
        tankAmmoAnimator      .paint ( component, graphics );
    }
}
[...]

```

`paint()`方法只是简单地委托给 `ComponentPainter` 实例。我们应该记得 `ComponentAnimator` 扩展了接口 `ComponentPainter`。

```

private void updateAnimators ( JComponent component )
    //////////////////////////////////////

```

```

{
    WorldAccessor worldAccessor = gameAccessor.getWorldAccessor ( );

    worldAnimator = new WorldAnimator (
        worldAccessor, audioClipCache, imageCache );

    TankAccessor playerTankAccessor
        = gameAccessor.getPlayerTankAccessor ( );

    if ( playerTankAccessor != null )
    {
        pathAnimator = new PathAnimator (
            gameAccessor, playerTankAccessor.getRadius ( ) );

        try
        {
            tankAmmoAnimator = new TankAmmoAnimator (
                playerTankAccessor, imageCache, component );
        }
        catch ( IOException ex )
        {
            ex.printStackTrace ( );
        }
    }
    else
    {
        pathAnimator = null;

        tankAmmoAnimator = null;
    }
}

```

在游戏中的每一个新关开始的位置，`updateAnimators()`方法都会创建委托 `ComponentAnimators` `worldAnimator`、`pathAnimator` 和 `tankAmmoAnimator` 的新实例。如果每一关的声音和图像都不相同，这个方法也会清除 `audioClipCache` 和 `imageCache`。

7.5.3 复合控制器

复合控制器包含并管理委托控制器。复合控制器对于根据游戏状态而过滤委托给控制器的事件来说非常有用。它们也可以用来启用或禁用多组委托控制器。复合控制器的示例并没有包含在 Mars 游戏中。这个游戏中的多个单独的控制器(例如 `SoundController` 和 `TankController`)都是独立操作的。

7.5.4 将三者进行组合

既然已经创建了各个模型、视图和控制器组件，那么就需要一个中心类将它们集中在一起创建游戏。从理论上讲，应该能从组件中混合和匹配所需要的东西，从而使创建新游戏工作变得简单。这将在后面章节中介绍多玩家联网模式时进行介绍。


```

package com.croftsoft.apps.mars;

[...]

import com.croftsoft.core.animation.*;
import com.croftsoft.core.animation.animator.*;
import com.croftsoft.core.animation.clock.Timekeeper;
import com.croftsoft.core.io.SerializableLib;
import com.croftsoft.core.media.sound.AudioClipCache;
import com.croftsoft.apps.mars.controller.FrameRateController;
import com.croftsoft.apps.mars.controller.SoundController;
import com.croftsoft.apps.mars.controller.TankController;
import com.croftsoft.apps.mars.controller.TimeController;
import com.croftsoft.apps.mars.model.Game;
import com.croftsoft.apps.mars.model.seri.SeriGame;
import com.croftsoft.apps.mars.view.GameAnimator;

public final class Main
    extends AnimatedApplet
    //////////////////////////////////////
    //////////////////////////////////////
{

[...]

private UserData userData;

private Game    game;

[...]

public static void main ( String [ ] args )
    //////////////////////////////////////
{
    launch ( new Main ( ) );
}

[...]

public Main ( )
    //////////////////////////////////////
{
    super ( createAnimationInit ( ) );
}

```

Main 类将模型、视图和控制器组合在一起。在这种情况下，Main 类将只知道一个模型(即 Game 模型)和一个视图(即 GameAnimator 视图)，因为它们是最上层联合。Main 扩展了 AnimatedApplet，并提供了一个静态的 main()方法，这样 AnimatedApplet 就可以作为 applet 或桌面应用程序进行部署了。

```

public void init ( )
    //////////////////////////////////////
{

```

```

super.init ( );

// persistent data

try
{
    userData = ( UserData ) SerializableLib.load'(
        LATEST_FILENAME,
        BACKUP_FILENAME,
        FILE_CONTENTS_SPEC,
        ( Applet ) this,
        PERSISTENCE_KEY,
        getClass ( ).getClassLoader ( ),
        RESOURCE_PATH_FILENAME );
}
catch ( Exception ex )
{
    ex.printStackTrace ( );
}

if ( userData == null )
{
    userData = new UserData ( );
}

```

init()方法首先调用其超类的 init(), 然后再加载持久的用户数据, 例如用户的个人喜好——声音是否应该关闭或打开。

```
game = new SeriGame ( );
```

使用接口引用创建并存储这个复合模型。

```

GameAnimator gameAnimator = new GameAnimator (
    game,
    animatedComponent,
    getClass ( ).getClassLoader ( ),
    MEDIA_DIR,
    BACKGROUND_COLOR );

addComponentAnimator ( gameAnimator );
AudioClipCache audioClipCache
    = gameAnimator.getAudioClipCache ( );

audioClipCache.setMuted ( userData.isMuted ( ) );

FrameRateAnimator frameRateAnimator
    = new FrameRateAnimator ( animatedComponent );

frameRateAnimator.toggle ( );

addComponentAnimator ( frameRateAnimator );

```


这里创建了视图的实例。FrameRateAnimator 本应该作为一个委托视图放置在 GameAnimator 的内部，但是我将它放置在 Main 中。我想也就只是在这个版本中才这样，因为现在只是用它来开发测试程序。

```

new FrameRateController (
    frameRateAnimator,
    animatedComponent );

new GameAnimatorController (
    gameAnimator,
    animatedComponent );

new TimeController (
    game.getTimekeeper ( ),
    game.getTimeFactorDefault ( ),
    TIME_FACTOR_DELTA,
    animatedComponent );

new SoundController (
    audioClipCache,
    userData,
    animatedComponent );

new TankController (
    game.getPlayerTank ( ).getTankOperator ( ),
    animatedComponent );
}

```

init()方法的最后一个任务就是创建控制器。animatedComponent 是以控制器构造函数的一个参数的形式传递进来的，这样它们就可以将它们自己与键盘和鼠标事件侦听器联系在一起。userData 实例也被传递到 SoundController 构造函数中，这样它就可以使声音功能的切换具有持久性。

```

public void update ( JComponent component )
///////////////////////////////////////////////////
{
    game.update ( );

    super.update ( component );
}

```

Main 扩展了 AnimatedApplet，后者实现了接口 ComponentUpdater。重写 update(JComponent) 方法以驱动 Game 的 update()方法。这就意味着，在这个实现中，游戏循环将会和动画循环联系在一起。在介绍“多玩家联网模式”时，这两个循环将会单独运行。

```

public void destroy ( )
///////////////////////////////////////////////////
{
    try
    {

```

```

        SerializableLib.save (
            userData,
            LATEST_FILENAME,
            BACKUP_FILENAME,
            FILE_CONTENTS_SPEC,
            ( Applet ) this,
            PERSISTENCE_KEY );
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }

    super.destroy ( );
}

```

destroy()方法保存修改以后的 userData，然后再委托给其超类的实现。

到此为止，我们已经将示例游戏 Mars 的源代码学习完毕。您可能希望将这一体系结构作为自己游戏的一个模板，尤其是当您的体系结构需要具备可伸缩性的时候。如果您希望只是像温习功课一样研究一下这个程序，可以考虑向这个游戏中添加新的实体类型，以便增加模型之间的交互。

7.6 数据驱动设计

在我看来，最好的体系结构就是一个可重用的体系结构。游戏引擎就是一个高可重用的体系结构，在这种体系结构中，游戏几乎完全是由事件驱动的。我们可以通过添加新内容开发新的游戏，而需定制的代码的数量也被最小化。

代码和内容之间到底有什么不同？人们一般会认为计算机穿孔卡是代码，而音乐卷带盘是内容，虽然它们都是在纸片上打一些小洞。可以将音乐文件看作代码，因为它提供了一个可以导致扬声器震动的指令序列。压缩的图像文件是点亮屏幕像素的一套硬编码的指令。在虚拟游戏世界中，带有怪物数据的一个游戏关地图，就是建立有趣的交互的一个方法。

在我看来，内容和代码之间惟一的区别就是，内容是由内容的创作者创建的，而代码是由代码编写人员创建的。这说明，二者真正区别是内容创作工具对游戏或游戏引擎是否可用。可以提供图形编辑器来创建图形，可以使用游戏关编辑器来创建关地图。一旦提供了这些工具，就已经实现了数据驱动设计。

脚本在内容和形式之间似乎很重要。我知道，过去用其他编程语言编写的游戏曾经使用 Java 作为它的脚本语言。那么如果游戏的主要组件也是用 Java 编写时，又该怎样考虑呢？也许应该将脚本分类为代码，也应该为内容创作人员的具体动作提供更简单的工具。

7.6.1 AnimationInit

通过创建一个名为 AnimationInit(在 com.croftsoft.core.animation 包中)的类，就开始了数据驱动的设计。AnimationInit 是一个数据对象类，该类一般存储定制游戏常用的一些数据参数，例如背景颜色、窗口大小、帧标题等。ArrayComponentUpdater 和 ArrayComponentPainter 存储

动画对象。尽管仍然在开发和测试阶段，但是还是使用了在第 6 章介绍的 JavaBeans XML 编码技术，提供了将 AnimationInit 实例保存到 XML 文件和从 XML 文件中加载 AnimationInit 实例的静态方法。理论上讲，使用 XML 数据编辑器而不使用任何其他代码能够创建一个独特的游戏。

7.6.2 AnimatedApplet

同一个包中的类 AnimatedApplet 使用了一个 AnimationInit 数据对象来初始化它自身。大部分游戏公共的代码都被封装在 AnimatedApplet 中，而游戏特定的内容(或代码)则都是由 AnimationInit 提供。在第 1 章中，我将 AnimatedApplet 作为 BasicsExample 的超类进行过简要介绍。现在当我开始开发新游戏的时候，它都是最理想的起点。

```
package com.croftsoft.core.animation;

[...]

public class AnimatedApplet
    extends JApplet
    implements ComponentAnimator, Lifecycle
    //////////////////////////////////////
    //////////////////////////////////////
{

    protected final AnimationInit      animationInit;

    protected final AnimatedComponent animatedComponent;

    //

    protected ArrayComponentUpdater arrayComponentUpdater;

    protected ArrayComponentPainter arrayComponentPainter;
```

AnimatedApplet 包含一个 animatedComponent，AnimatedApplet 也是驱动 animatedComponent 的那个 ComponentAnimator。一般情况下，AnimatedApplet 都通过委托给 arrayComponentUpdater 和 arrayComponentPainter，作为 ComponentAnimator 执行它的职责，而 arrayComponentUpdater 和 arrayComponentPainter 可能是由 animationInit 数据对象提供的。

```
public static void main ( String [ ] args )
    throws Exception
    //////////////////////////////////////
{
    AnimationInit animationInit = null;

    if ( ( args == null )
        || ( args.length < 1 ) )
    {
        animationInit = new AnimationInit ( );
    }
    else
```

```

    {
        animationInit = AnimationInit.load ( args [ 0 ] );
    }

    launch ( new AnimatedApplet ( animationInit ) );
}

```

与其编写一个 `AnimatedApplet` 的子类提供 `animationInit` 数据对象，还不如直接以命令行参数的形式提供一个 `AnimationInit` 的 XML 数据文件，直接启动 `AnimatedApplet` 类的一个实例。

```

public static void launch ( AnimatedApplet animatedApplet )
///////////////////////////////////////////////////
{
    AnimationInit animationInit = animatedApplet.animationInit;

    LifecycleWindowListener.launchAppletAsDesktopApp (
        animatedApplet,
        animationInit.getFrameTitle ( ),
        animationInit.getFrameIconFilename ( ),
        animatedApplet.getClass ( ).getClassLoader ( ),
        true, // useFullScreenToggler,
        animationInit.getFrameSize ( ),
        animationInit.getShutdownConfirmationPrompt ( ) );
}

```

可以使用静态方法 `launch()` 以桌面应用程序的方式启动一个 `animatedApplet`。帧参数是从关联的 `animationInit` 中获取的。

```

public AnimatedApplet ( AnimationInit animationInit )
///////////////////////////////////////////////////
{
    NullPointerException.check ( this.animationInit = animationInit );

    Double frameRate = animationInit.getFrameRate ( );

    if ( frameRate == null )
    {
        animatedComponent = new AnimatedComponent ( this );
    }
    else
    {
        animatedComponent
            = new AnimatedComponent ( this, frameRate.doubleValue ( ) );
    }
}

```

主构造函数带有一个 `animationInit` 参数。像在 `BasicsExample` 中采取的方法一样，这也可以由子类的构造函数提供。

```

public AnimatedApplet ( )
///////////////////////////////////////////////////
{

```



```

    this ( new AnimationInit ( ) );
}

```

无参数的构造函数使用一个带有默认值的 AnimationInit 实例。

```

public String getAppletInfo ( )
///////////////////////////////////////////////////////////////////
{
    return animationInit.getAppletInfo ( );
}

public void init ( )
///////////////////////////////////////////////////////////////////
{
    String appletInfo = getAppletInfo ( );

    if ( appletInfo != null )
    {
        System.out.println ( appletInfo );
    }

    Color backgroundColor = animationInit.getBackgroundColor ( );

    if ( backgroundColor != null )
    {
        animatedComponent.setBackground ( backgroundColor );
    }

    Color foregroundColor = animationInit.getForegroundColor ( );

    if ( foregroundColor != null )
    {
        animatedComponent.setForeground ( foregroundColor );
    }

    Font font = animationInit.getFont ( );

    if ( font != null )
    {
        animatedComponent.setFont ( font );
    }

    Cursor cursor = animationInit.getCursor ( );

    if ( cursor != null )
    {
        animatedComponent.setCursor ( cursor );
    }

    arrayComponentUpdater = animationInit.getArrayComponentUpdater ( );

    arrayComponentPainter = animationInit.getArrayComponentPainter ( );
}

```

```

    if ( arrayComponentUpdater == null )
    {
        arrayComponentUpdater = new ArrayComponentUpdater ( );
    }

    if ( arrayComponentPainter == null )
    {
        arrayComponentPainter = new ArrayComponentPainter ( );
    }

    Container contentPane = getContentPane ( );

    contentPane.setLayout ( new BorderLayout ( ) );

    contentPane.add ( animatedComponent, BorderLayout.CENTER );

    animatedComponent.init ( );

    validate ( );
}

```

init()方法使用 animationInit 数据对象中的数据初始化这个 applet。

```

public void start    ( ) { animatedComponent.start    ( ); }
public void stop     ( ) { animatedComponent.stop     ( ); }
public void destroy  ( ) { animatedComponent.destroy  ( ); }

```

其他生命周期方法只是简单地委托给animatedComponent。

```

public void update ( JComponent component )
///////////////////////////////////////////////////////////////////
{
    arrayComponentUpdater.update ( component );
}

public void paint (
    JComponent component,
    Graphics2D graphics )
///////////////////////////////////////////////////////////////////
{
    arrayComponentPainter.paint ( component, graphics );
}

```

方法 update()和 paint()可以被子类重写。

```

public void addComponentAnimator (
    ComponentAnimator componentAnimator )
///////////////////////////////////////////////////////////////////
{
    addComponentUpdater ( componentAnimator );
}

```



```

        addComponentPainter ( componentAnimator );
    }

    public void addComponentPainter (
        ComponentPainter componentPainter )
    ////////////////////////////////////////////
    {
        arrayComponentPainter.add ( componentPainter );
    }

    public void addComponentUpdater (
        ComponentUpdater componentUpdater )
    ////////////////////////////////////////////
    {
        arrayComponentUpdater.add ( componentUpdater );
    }

```

然而，一般情况下，我们最想要的行为就是分别委托给 `arrayComponentUpdater` 和 `arrayComponentPainter` 的 `update()`方法和 `paint()`方法的默认行为。使用上面列出的便利方法，可以将其他一些动画组件在初始化以后添加到这些数组中。

7.7 小结

介绍游戏体系结构的这一章可以归纳为下面这些设计规则：

- 尽可能使用接口引用代替具体类的引用。
- 避免白盒继承，并使用通过接口组合的设计来代替白盒引用。
- 将游戏分解为相互独立的模型、视图和控制器组件。
- 使用一个虚拟世界类来创建、存储和检索游戏的实体模型。
- 将游戏循环的模型更新阶段分为单独的准备和更新这两步。
- 正常的控制器和模型交互作用的外部逻辑应归于游戏类。
- 游戏没有视图和控制器就不可以自己运行。
- 游戏世界视图只为游戏世界里的游戏实体模型提供视图。
- 在用户界面屏幕上的所有其他对象都由游戏视图提供。
- `Main` 类将模型、视图和控制器装配到一个可运行的游戏中。
- 数据驱动的体系结构是高可重用的。

7.8 参考文献

Grand, Mark. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, 2nd edition, Vol. 1. Hoboken, NJ: John Wiley & Sons, 2002.

Gamma, Erich et al. "How Design Patterns Solve Design Problems." Section 1.6 in *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley Publishing, 1995.

Rollings, Andrew and Dave Morris. "Game Architecture." Part III in *Game Architecture and Design*. Scottsdale, AZ: Coriolis Technology Press, 2000.

第 8 章 A* 算法

实践中的绝大部分知识都没有很伟大的使用价值。

——本杰明·富兰克林

如果将人们在人工智能(Artificial Intelligence, AI)领域所作的学术研究与在游戏编程行业里的实际实践相比较,将会发现这二者基本上没有交集。对其中大部分游戏而言,绝大部分程序员只会去使用对他们游戏 AI 有利的东西,即使这会意味着可能要编写多个简单的 if/then 语句。与其集成一个基于规则的专家系统的外壳(例如 Jess),在实践中人们更愿意对游戏采用这种更流行的方法:使用发布-订阅(publish and subscribe)体系结构处理状态更新事件¹,再使用 if/then 语句决定一个参与者是否应该对收到的一个具体的事件作出反应,以及怎样作出反应。

在代码库和演示程序内部,您会发现一些使用了神经网络和遗传算法的示例。这些示例专门研究格斗或逃走(fight-or-flight)策略,或在敌对环境中研究最佳可能的移动方位。如果希望修改这些研究技术,应该对在游戏编程行业内得到广泛使用的一项 AI 理论方法了如指掌,这个理论就是 A*算法。

A*算法通常用在与绕过障碍物进行路径发现相关的游戏中,但是它也可以应用在其他类型的需要启发式状态空间搜索的游戏中,例如国际象棋游戏。在更加简单的一些游戏中(例如 tic-tac-toe 游戏)也不经常使用这个算法,因为这种游戏中移动的数目很少,这种游戏中的移动,只需使用适当数目的 if/then 语句或查找表进行静态编码,就可以处理了。当从起点到目标的可能路径数目相对于合理的处理时间而言,预计算的计算量太多或要耗尽几乎所有资源的情况下,A*算法最为实用。

A*算法的基本思想就是计算怎样“从这里到那里”的问题,这里的“那里”就是虚拟空间中的一个位置或成功的移动。该算法计算怎样“从这里到那里”问题的基本方法是:通过考虑需要注意的事项,从当前的位置开始,在每一个方向上都看一步或向前移动,然后再估计到达目标的距离。那些看起来最有希望的路径会被首先探测。如果遇到了一个障碍物或死胡同,就再检查具有较小希望的直线路径。这种搜索处理连续进行,一次一步,直到到达目标为止。

图 8-1 给出了实际使用中 A*算法的一个实现。规划好的路径以红色圆环的形式标注在视图上。在这个场景中,右上角的坦克正在规划一个绕过附近障碍物的行驶路径,以到达左下角的一个空坦克的位置。可以通过按下 Mars 游戏中的选项键 P 来跟踪动画的路径投影,从而观察行动过程中规划的动态路径。在最后用户输入以后大约 15 秒,玩家坦克就会启动自动驾驶模式,A*算法的演示也是从这里开始的。

1 <http://herzberg.ca.sandia.gov/jess/>

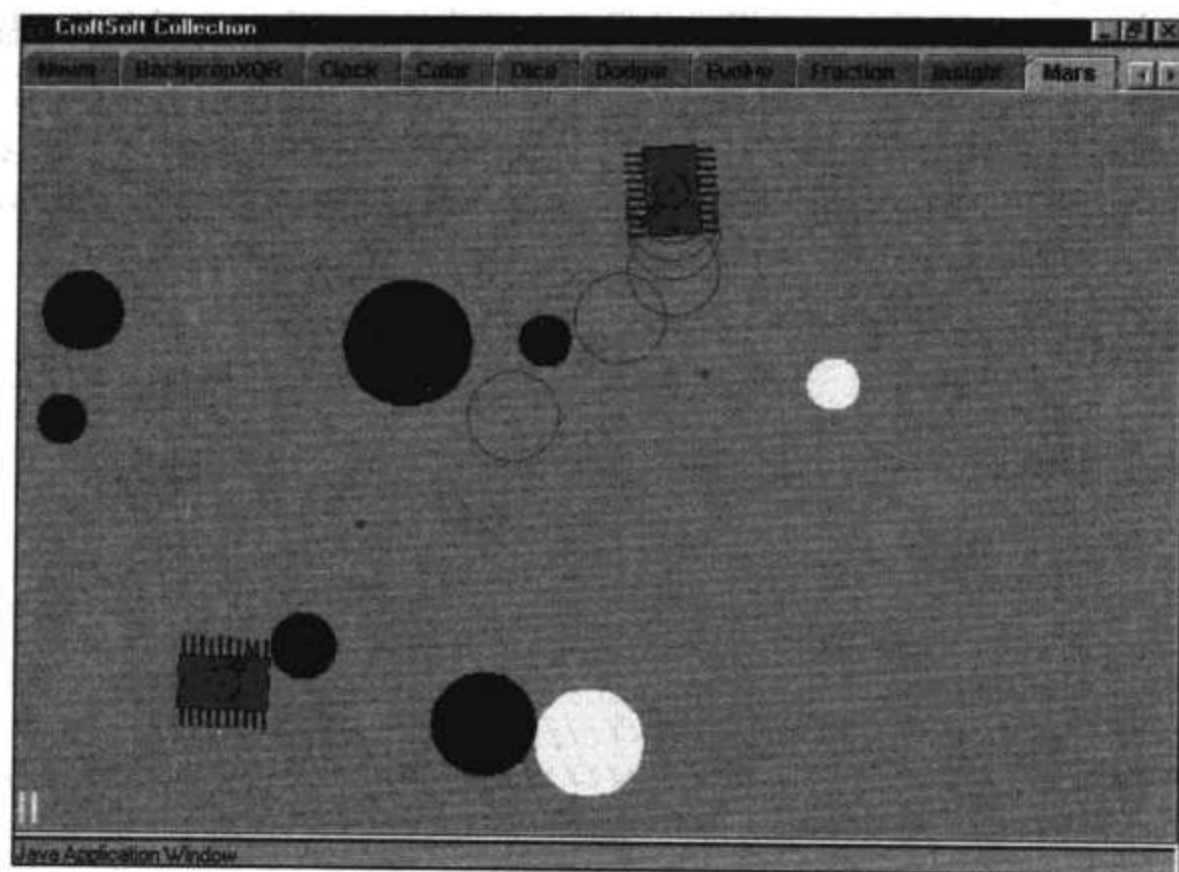


图 8-1 路径投影

在 Mars 示例中使用的 A*算法的实现被设计为高可重用的，这样就可以在您自己的游戏中使用该实现。下面将详细介绍可重用 A*算法包中的这些类，以及是怎样将它们集成到 Mars 游戏中的。

8.1 Cartographer

```
package com.croftsoft.core.ai.astar;

import java.util.Iterator;

public interface Cartographer
////////////////////////////////////
////////////////////////////////////
{

    public double estimateCostToGoal ( Object node );

    public Iterator getAdjacentNodes ( Object node );

    public double getCostToAdjacentNode (
        Object fromNode,
        Object toNode );

    public boolean isGoalNode ( Object node );
```

为了能够使 A*算法的实现尽可能被重用，就需要将游戏特定的数据和逻辑分隔开来。com.croftsoft.core.ai.astar 包中接口 Cartographer 的一个实现映射了虚拟的世界。例如，它知道从一个给定的位置出发，可以从除了东以外的其他所有方向进入，因为向东的道路是封闭的。

在国际象棋游戏中, Cartographer 可以提供从棋盘上当前状态可得出的所有合法移动, 以及关于该移动是沿着哪一个方向的等相关信息。它可以告诉您在战车翻越一座山和绕过这座山在代价上的区别。它也可以估算必须要走多远, 并可以告诉您到达终点的时间。当有多个目标的时候(例如寻找所有可用的燃料补给站), Cartographer 都会将您带到最近的那一条路径上。

8.2 NodeInfo

```
package com.croftsoft.core.ai.astar;

import com.croftsoft.core.lang.NullArgumentException;

public final class NodeInfo
    implements Comparable
    //////////////////////////////////////
    //////////////////////////////////////
{

    private final Object node;

    //

    private double costFromStart;

    private NodeInfo parentNodeInfo;

    private double totalCost;
```

为了知道一个路径是否比另一个路径更好, 需要作一些比较。如果在预算上作一些调整, 就可能要计算出哪一个路径会使从开始到结束所花的钱最少。但是如果对花费的时间很在意, 可能还需要将采取低速传输方式(例如船舶与飞机)的代价考虑进去。在花费的代价方面, 怎样考虑这些变量将决定最终选择的路径。

当面临众多选择的时候, 为每一个可能的路径计算从起点到终点的代价可能不太现实。对状态空间中一个给定的节点(node)而言, 当一次一步地从一个节点跳到下一个节点的时候, 您将会知道 costFromStart, 因为必须检查沿途的障碍物。要探测出哪一个邻近的节点是基于从起点到终点的这个 totalCost 的, 而 totalCost 是可以估算的, 因为 totalCost 就是已知的 costFromStart 再加上从该点到目标的估计花费。

注意, 这个 totalCost 只是一个估计值。进一步的检查可能会检查出从这一点出发的路径的代价更加高, 或者甚至是完全封闭的。只有在向目标的搜索空间中执行了一步一步的移动以后, 才可以真正地知道从起点到目标移动的真正代价。

当然, 找到了一条到达目标的路径以后, 就需要记住从起始位置到达目标位置的这个过程所采取的步骤。变量 parentNodeInfo 为此目的提供了一个到前一步的链接。

```
public NodeInfo ( Object node )
    //////////////////////////////////////
{
    NullArgumentException.check ( this.node = node );
```



```

    }

    [...]

    public int compareTo ( Object other )
    //////////////////////////////////////
    {
        NodeInfo otherNodeInfo = ( NodeInfo ) other;

        double otherTotalCost = otherNodeInfo.totalCost;

        if ( totalCost < otherTotalCost )
        {
            return -1;
        }

        if ( totalCost > otherTotalCost )
        {
            return 1;
        }

        return 0;
    }

```

一旦有了估计的总代价，就可以在一部分路径和另一部分路径之间作一个合理的比较，并更进一步地检查最有可能首先采用的那一个路径。类 `NodeInfo` 实现了接口 `Comparable` 的方法 `compareTo()`，专门用来作这种比较。

8.3 AStar

```

package com.croftsoft.core.ai.astar;

import java.util.*;

import com.croftsoft.core.lang.NullArgumentException;

public final class AStar
    //////////////////////////////////////
    //////////////////////////////////////
    {

        private final Cartographer cartographer;

        private final List openNodeInfoSortedList;

        private final Map nodeToNodeInfoMap;

```

A*算法保存了要进行探测节点的一个列表。这个列表(也即 `openNodeInfoSortedList`)是按最小代价进行排序的，这样，最有希望的路径最优先。当和邻近的节点一起出现的时候，这个节

点就可能在前面已经探测过了，因为到达任意一个给定节点的路径都可能不止一个。这可以通过检查是否已经存在一个与在 `nodeToNodeInfoMap` 中给出的节点关联的 `NodeInfo` 对象的方法进行判定。

```
private NodeInfo bestNodeInfo;
```

```
private double bestTotalCost;
```

```
private NodeInfo goalNodeInfo;
```

```
private boolean listEmpty;
```

在 AStar 搜索时，我们可以在任何时候中断它，并要求它进行报告。为此，它要维护信息 `bestNodeInfo`、`bestTotalCost`、`goalNodeInfo` 和 `listEmpty`。

```
public AStar ( Cartographer cartographer )
///////////////////////////////////////////////////
{
    NullPointerException.check ( this.cartographer = cartographer );

    nodeToNodeInfoMap = new HashMap ( );

    openNodeInfoSortedList = new LinkedList ( );
}

///////////////////////////////////////////////////
///////////////////////////////////////////////////

public boolean isGoalFound ( ) { return goalNodeInfo != null; }

public boolean isListEmpty ( ) { return listEmpty; }

public Iterator getPath ( )
///////////////////////////////////////////////////
{
    List pathList = new LinkedList ( );

    NodeInfo nodeInfo = goalNodeInfo;

    if ( nodeInfo == null )
    {
        nodeInfo = bestNodeInfo;
    }
    while ( nodeInfo != null )
    {
        NodeInfo parentNodeInfo = nodeInfo.getParentNodeInfo ( );

        if ( parentNodeInfo != null )
        {
            pathList.add ( 0, nodeInfo.getNode ( ) );
        }
    }
}
```



```

        nodeInfo = parentNodeInfo;
    }

    return pathList.iterator ( );
}

```

getPath()方法将使用 parentNodeInfo 链接从 goalNodeInfo 原路返回，以便从紧跟起点之后的第一步开始重新构造路径。再使用一个 Iterator 将这些节点再按前向顺序返回。注意，如果这种搜索在找到目标节点之前就已经中断，那么就会使用 bestNodeInfo 而不是使用 goalNodeInfo。该 bestNodeInfo 在搜索过程中到达那一点后，才会给出通往目标的最优猜测路径。但是 goalNodeInfo 中包含了具有最低实际 costFromStart 的节点，bestNodeInfo 将会包含直到那一点才被发现的具有最低估计的 totalCost。可以调用 isGoalFound()方法来决定返回的路径是要给 goalNodeInfo 还是要给 bestNodeInfo。

```

public Object getFirstStep ( )
///////////////////////////////////////////////////////////////////
{
    NodeInfo nodeInfo = goalNodeInfo;

    if ( nodeInfo == null )
    {
        nodeInfo = bestNodeInfo;
    }

    Object node = null;

    while ( nodeInfo != null )
    {
        NodeInfo parentNodeInfo = nodeInfo.getParentNodeInfo ( );

        if ( parentNodeInfo != null )
        {
            node = nodeInfo.getNode ( );
        }

        nodeInfo = parentNodeInfo;
    }

    return node;
}

```

有时不需要整个路径，仅仅需要第一个步骤，因为环境是连续变化的。在采取了第一步以后，就需要重新计算一次路径，因为障碍物可能已经移动，挡住了原来路径(前面计算以后的路径)的位置。

```

public void reset ( Object startNode )
///////////////////////////////////////////////////////////////////
{
    goalNodeInfo = null;
}

```

```

listEmpty = false;

openNodeInfoSortedList.clear ( );

nodeToNodeInfoMap.clear ( );

NodeInfo nodeInfo = new NodeInfo ( startNode );

nodeToNodeInfoMap.put ( startNode, nodeInfo );

openNodeInfoSortedList.add ( nodeInfo );

bestTotalCost = Double.POSITIVE_INFINITY;
}

```

如果环境中有些东西发生了变化，就需要重设搜索数据，即使出发的位置相同。但是，如果知道游戏中的障碍物没有移动，那么就可以预计算或缓存节点到节点的代价信息。

```

public boolean loop ( )
///////////////////////////////////////////////////////////////////
{
    if ( openNodeInfoSortedList.isEmpty ( ) )
    {

        listEmpty = true;

        return false;
    }
}

```

状态空间搜索可能会花较长的一段时间，这段时间通常比从给定的起始位置找到目标所需要的时间要长，在这段时间里仍然会维持原来理想的帧速率。正因为如此，您将希望限制在一帧中搜索节点的数目。调用 `loop()` 方法以便一次只搜索一个节点。如果循环应该连续，它就返回 `true`。如果找到了目标或没有其他要搜索的节点，它就返回 `false`。

```

NodeInfo nodeInfo
= ( NodeInfo ) openNodeInfoSortedList.remove ( 0 );

Object node = nodeInfo.getNode ( );
if ( cartographer.isGoalNode ( node ) )
{
    if ( ( goalNodeInfo == null )
        || ( nodeInfo.getCostFromStart ( )
            < goalNodeInfo.getCostFromStart ( ) ) )
    {
        goalNodeInfo = nodeInfo;
    }

    return false;
}

```


总代价的估计值最小的节点会从列表中弹出，并检查该节点是否就是目标节点。如果是目标节点，它就会替换前面发现的作为 `goalNodeInfo` 的目标节点(如果这个 `costFromStart` 更小的话)。当游戏地图有多个目标和快捷路径，例如超时空转移，那么就会发生这种情况。由于超时空转移的跳点之间移动的花费为 0 或花费大大降低，所以对从这样的一个起始点到达目标所花费代价的所有估计值，使用直线距离的测量值都会过高。这样优先顺序列表中的其他节点会先被搜索，就会导致具有较高 `costFromStart` 的节点先被发现。

可以通过不高估到目标节点的花费来避开这个问题。保证这一点的一个简单方法就是，即使是在有超时空转移的游戏地图里，也总是让 `estimateCostToGoal()` 方法返回一个值，例如 0.0。在这种情况下，估计的 `totalCost` 就会等于这个已知的 `costFromStart` 加上 0 值，没有探测过节点的列表的优先顺序就会只基于 `costFromStart` 一个值。这保证了找到的第一个目标节点是具有最低 `costFromStart` 的节点。但是遗憾的是，当这样实现的时候，大部分环境中的搜索将会慢一些——相对于已经使用了某种估计指导规则在状态空间进行的搜索而言。

理想的情况是，游戏地图中并没有像超时空转移这种怪异的跳点，`estimateCostToGoal()` 的功能永远不会产生实际花费的过高估计。在这种情况下，可以得到这样的保证：状态空间中不会有另外一个目标节点的 `costFromStart` 比通过 A* 算法发现的第一个节点的 `costFromStart` 低。如果在状态空间中只有一个目标，也可以得到这样的保证：A* 算法将会返回通向它且具有最小 `costFromStart` 的路径。在这些环境中，在第一个目标节点被发现以后，就没有必要再连续进行 A* 算法的循环以查找一个更好的路径了。

```
Iterator iterator = cartographer.getAdjacentNodes ( node );
```

```
while ( iterator.hasNext ( ) )
{
    Object adjacentNode = iterator.next ( );
```

如果优先列表中的当前节点不是目标节点，搜索就会连续下去。就会调用 `Cartographer` 的 `getAdjacentNodes()` 方法取出与当前节点邻近的那些节点。然后也是一次一个地处理这些节点。

```
double newCostFromStart
    = nodeInfo.getCostFromStart ( )
    + cartographer.getCostToAdjacentNode ( node, adjacentNode );
```

邻近节点的 `newCostFromStart` 是到当前节点的 `costFromStart` 的花费加上从当前的节点到邻近节点的花费之和。与到目标的估计不同，这是从起始节点出发的实际花费，并且需要一次一步地仔细进行计算。

```
NodeInfo adjacentNodeInfo
    = ( NodeInfo ) nodeToNodeInfoMap.get ( adjacentNode );

if ( adjacentNodeInfo == null )
{
    adjacentNodeInfo = new NodeInfo ( adjacentNode );

    nodeToNodeInfoMap.put ( adjacentNode, adjacentNodeInfo );

    openNodeInfoSortedList.add ( adjacentNodeInfo );
}
```

如果该节点是第一次看到的邻近节点，就会被添加到未探测节点的列表 `openNodeInfoSortedList` 中。

```
else if (
    adjacentNodeInfo.getCostFromStart ( ) <= newCostFromStart )
{
    continue;
}
```

如果不是第一次发现这个邻近的节点，那么就可能还存在一条从起始节点(在搜索中较早发现的)到达它而且更短的路径。如果这种情况必须通过比较原来的 `costFromStart` 和 `newCostFromStart`，那么从当前节点到邻近节点的那条路径就会被认为不是最佳的路径，在搜索过程中也会不再对其进行追踪。

```
adjacentNodeInfo.setParentNodeInfo ( nodeInfo );

adjacentNodeInfo.setCostFromStart ( newCostFromStart );

double totalCost = newCostFromStart
    + cartographer.estimateCostToGoal ( adjacentNode );

adjacentNodeInfo.setTotalCost ( totalCost );
```

`adjacentNodeInfo` 被更新，要么是因为这是该节点第一次被探测，要么就是因为搜索发现了从当前节点到达该邻近节点的一条更短的路径。

```
if ( totalCost < bestTotalCost )
{
    bestNodeInfo = adjacentNodeInfo;

    bestTotalCost = totalCost;
}
```

如果从起始位置到目标估计的总代价比以前已经发现的代价要小，那么这个邻近的节点就会被认为是迄今为止已经发现的到达目标的最佳候选路径。

```
Collections.sort ( openNodeInfoSortedList );
}

return true;
}
```

由于该邻近节点是第一次被添加到这个列表中，或由于更新了该列表中节点的 `totalCost` 值，因此该列表需要重新进行排序。我认为使用默认 `Collections` 排序算法的排序操作将会相当快速，因为永远不会有一个以上元素的位置不恰当，在任何时候也只有一个 `adjacentNodeInfo` 被添加到具有优先次序的列表中，或只有一个 `adjacentNodeInfo` 被更新。

`loop()`方法最后返回 `true`，表示如果时间允许的话，它应该被再次调用以连续地搜索目标。

8.4 AStarTest

```
package com.croftsoft.core.ai.astar;

import java.awt.Point;
import java.util.*;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.Testable;

public final class AStarTest
    implements Cartographer, Testable
    //////////////////////////////////////
    //////////////////////////////////////
{
```

类 AStarTest 测试上面给出的 Astar 实现。它也作为 Cartographer 实现的一个最原始的示例。

```
private static final int MIN_X = -10;

private static final int MIN_Y = -10;

private static final int MAX_X = 10;

private static final int MAX_Y = 10;
```

在游戏世界上设置边界，这样当不能到达目标的时候，搜索就不会连续地无限进行下去。当目标(但不是起始点)完全被障碍物阻碍以后就会发生这种情形。在这种情况下，只要游戏世界上还有节点没有被探测到，就会一直进行无尽地搜索。不会返回在这些最小值和最大值边界限制以外的其他邻近节点。

```
private final Set blockedSet;

private final Point goalPoint;

private final Point jumpPoint;
```

AStarTest 使用 java.awt.Point 类型作为基本的状态空间节点类型。这同时也暗示了它是一个二维的瓦片世界，因为 Point 存储整型的 x 和 y 坐标。障碍物(例如墙)是以 blockedSet 中 Point 元素的形式存储的。这里的 jumpPoint(如果有的话)，就是一个特殊的超时空转移的跳点，无论这个点离 goalPoint 有多远，它到达 goalPoint 的花费总是为 0。这用来测试这样的 Astar：这种 Astar 带有一个特殊的世界地图，其中两点之间最短的路径不是第一条被发现的路径。

```
public static void main ( String [ ] args )
    //////////////////////////////////////
{
    System.out.println ( test ( args ) );
}
```

```

public static boolean test ( String [ ] args )
///////////////////////////////////////////////////////////////////
{
    try
    {
        // Finds its way around a wall.

        AStarTest aStarTest1 = new AStarTest (
            new Point ( 4, 0 ),
            new Point [ ] {
                new Point ( 1, 1 ),
                new Point ( 1, 0 ),
                new Point ( 1, -1 ) } );

        [...]

        // Goal clear on right but teleport jump on left closer.

        AStarTest aStarTest5 = new AStarTest (
            new Point ( MAX_X, 0 ),
            new Point [ 0 ],
            new Point ( -3, 0 ) );

        return test ( aStarTest1 )
        [...]
        && test ( aStarTest5 );
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );

        return false;
    }
}

```

测试包括找到一条绕过障碍墙到达目标的一条路径，当起点或目标点完全被墙围起来以后会终止，使用超时空转移的跳点到达一个被障碍墙围起来的目标，找到到达目标的一条直通路，以及最后通过与超时空转移相反的方向找到一条更短的路径。

```

public static boolean test ( AStarTest aStarTest )
///////////////////////////////////////////////////////////////////
{
    System.out.println ( "Testing..." );

    AStar aStar = new AStar ( aStarTest );

    aStar.reset ( new Point ( 0, 0 ) );

    while ( true )
    {
        aStar.loop ( );
    }
}

```



```

        if ( aStar.isListEmpty ( ) )
        {
            break;
        }
    }

    System.out.println ( "goalFound: " + aStar.isGoalFound ( ) );

    Iterator iterator = aStar.getPath ( );

    while ( iterator.hasNext ( ) )
    {
        System.out.println ( iterator.next ( ) );
    }

    return true;
}

```

这个 `test()` 方法将会连续循环，直到所有可能的节点都被探测一遍。然后它就会报告是否已经发现目标，并显示出相应的路径。注意，一般来讲，当找到第一个目标的时候，或达到某些循环的最大次数以后，就会退出该循环。在这个方法中执行一个穷尽式搜索，以达到测试使用超时空转移的跳点发现更短路径的目的。

```

public double estimateCostToGoal ( Object node )
///////////////////////////////////////////////////
{
    return getCostToAdjacentNode ( node, goalPoint );
}

```

`estimateCostToGoal()` 方法只是简单地委托给 `getCostToAdjacentNode()`，其方法是将 `goalPoint` 用作参数 `toNode`。

```

public Iterator getAdjacentNodes ( Object node )
///////////////////////////////////////////////////
{
    Point nodePoint = ( Point ) node;

    int x = nodePoint.x;

    int y = nodePoint.y;

    List list = new ArrayList ( );

    if ( nodePoint.equals ( jumpPoint ) )
    {
        list.add ( goalPoint );

        return list.iterator ( );
    }

    for ( int offsetX = -1; offsetX < 2; offsetX++ )

```

```

{
    for ( int offsetY = -1; offsetY < 2; offsetY++ )
    {
        if ( ( offsetX == 0 )
            && ( offsetY == 0 ) )
        {
            continue;
        }

        int newX = x + offsetX;

        int newY = y + offsetY;

        if ( ( newX < MIN_X )
            || ( newY < MIN_Y )
            || ( newX > MAX_X )
            || ( newY > MAX_Y ) )
        {
            continue;
        }

        Point point = new Point ( newX, newY );

        if ( !blockedSet.contains ( point ) )
        {
            list.add ( point );
        }
    }

    return list.iterator ( );
}

```

方法 `getAdjacentNodes()` 只是简单地返回 8 个主要方向上邻近的瓦片，除非它们被墙阻塞或在游戏世界的边界以外。如果当前的节点是超时空转移的 `jumpPoint`，那么 `goalPoint` 就会以邻近节点的身份返回。

```

public double getCostToAdjacentNode (
    Object fromNode,
    Object toNode )
{
    ///////////////////////////////////////////////////////////////////
    if ( fromNode.equals ( jumpPoint ) )
    {
        return 0.0;
    }

    return ( ( Point ) fromNode ).distance ( ( Point ) toNode );
}

```

如果 `fromNode` 不是超时空转移的 `jumpPoint`，那么到达邻近节点的代价就只是两点之间的距离。

如果使用 A*算法玩一个国际象棋的游戏而不是用来移动虚拟世界中的实体，那么从起始位置出发的代价就可以在移动的数目中进行度量，根据一些规则获得假想的棋盘状态。然后，估算到达目标的代价可能就是从该位置上成功移动数目的一个判断，该判断基于棋盘上其余棋子的相对兵力。我并不是说 A*算法是用于象棋游戏的最好算法，但是我确实希望实现这一点：A*算法是非常通用的，就像启发式状态空间搜索算法一样。

```
public boolean isGoalNode ( Object node )
////////////////////////////////////
{
    return ( ( Point ) node ).equals ( goalPoint );
}
```

为了进行测试，isGoalNode()方法使用了一个相等性检查代替一致性检查，因为邻近节点的新实例是在 getAdjacentNodes()方法中创建的。一个最佳的方法可能就是当游戏启动的时候，预先分配所有搜索节点的对象，这样就没有必要在每次调用 getAdjacentNodes()方法的时候，都必须重新创建一次。当状态空间中可能存在的总节点数目的大小比较合理的时候，您可能希望在每次调用 getAdjacentNodes()方法时，都重新创建一个搜索节点的对象。

您可能也会遇到状态空间中节点的数量较小，但是它们之间可能路径的数目却很大的情况。例如，假设每一个节点都和几乎所有其他节点相通就是这种情况。假设有 3 个节点，那么 A 节点和 C 节点之间的可能路径就是两个 A-C 和 A-B-C。假设有 4 个节点，那么 A 节点和 D 节点之间的路径的数目可能就有 5 个：A-D、A-B-D、A-B-C-D、A-C-D 和 A-C-B-D。假设有 5 个节点，那么路径的数目就比较大了。在这种情况下，即使路径的数目很多，状态空间节点的数目很少，可以为所有节点对象分配空间，也不能完全探测，而必须使用启发式搜索进行遍历。

8.5 SpaceTester

```
package com.croftsoft.core.ai.astar;

import com.croftsoft.core.math.geom.PointXY;

public interface SpaceTester
////////////////////////////////////
////////////////////////////////////
{

    public boolean isSpaceAvailable ( PointXY pointXY );
}
```

接口 SpaceTester 定义了一个方法，该方法测试二维真实空间中的一个点是否有像墙这样的障碍物。如果节点落在游戏世界的边界以外，它也可以返回 false。

8.6 GridCartographer

```
package com.croftsoft.core.ai.astar;

import java.util.*;
```

```

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.math.geom.Point2DD;
import com.croftsoft.core.math.geom.PointXY;

public final class GridCartographer
    implements Cartographer
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

```

GridCartographer 是 Cartographer 实现的一个示例，Cartographer 的这个实现是专门为 2D 世界定制的，并带有浮点坐标。它使用 SpaceTester 的一个实现来判定某一个节点是空旷的还是障碍物的。这样就使 GridCartographer 具有高可重用性。

```

public static final double DEFAULT_STEP_SIZE = 1.0;

```

没有用整型坐标分隔成多个瓦片块的真实游戏世界给 A* 算法带来了特殊的挑战，因为其中有无数个可能的状态空间搜索节点。GridCartographer 通过在两个邻近的节点之间叠加一个 DEFAULT_STEP_SIZE 为 1.0 的一个矩形网格，可以解决这个问题。

```

private final List adjacentList;

private final double stepSize;

private final SpaceTester spaceTester;

//

private PointXY goalPointXY;

///////////////////////////////////////////////////
///////////////////////////////////////////////////

public GridCartographer (
    SpaceTester spaceTester,
    double stepSize )
{
    ///////////////////////////////////////////////////
    NullArgumentException.check ( this.spaceTester = spaceTester );

    this.stepSize = stepSize;

    adjacentList = new ArrayList ( );
}

public GridCartographer ( SpaceTester spaceTester )
{
    ///////////////////////////////////////////////////
    this (
        spaceTester,
        DEFAULT_STEP_SIZE );
}

```


主构造函数允许将 `stepSize` 设为您选择的任何值，例如 0.001 或 40.0。当 `stepSize` 较小的时候，A*算法在找到一条到达目标的直线距离方面能够做得更好一些，但是在具体的处理中，它也需要花更长的时间，并消耗更多的内存。

```
public void setGoalPointXY ( PointXY goalPointXY )
////////////////////////////////////////////////////
{
    NullPointerException.check ( this.goalPointXY = goalPointXY );
}
////////////////////////////////////////////////////
// interface Cartographer methods
////////////////////////////////////////////////////

public double estimateCostToGoal ( Object node )
////////////////////////////////////////////////////
{
    return ( ( PointXY ) node ).distanceXY ( goalPointXY );
}

public Iterator getAdjacentNodes ( Object node )
////////////////////////////////////////////////////
{
    PointXY pointXY = ( PointXY ) node;

    adjacentList.clear ( );

    double distanceToGoal = pointXY.distanceXY ( goalPointXY );

    if ( distanceToGoal <= stepSize )
    {
        adjacentList.add ( goalPointXY );

        return adjacentList.iterator ( );
    }

    double x = pointXY.getX ( );

    double y = pointXY.getY ( );

    for ( int ix = -1; ix < 2; ix++ )
    {
        for ( int iy = -1; iy < 2; iy++ )
        {
            if ( ( ix == 0 )
                && ( iy == 0 ) )
            {
                continue;
            }

            PointXY step = new Point2DD (
                ( ( int ) ( x / stepSize ) + ix ) * stepSize,
```

```

        ( ( int ) ( y / stepSize ) + iy ) * stepSize );
    if ( spaceTester.isSpaceAvailable ( step ) )
    {
        adjacentList.add ( step );
    }
}

return adjacentList.iterator ( );
}

```

当 `distanceToGoal` 小于或等于 `stepSize` 的时候，就将 `goalPointXY` 作为邻近的节点返回。这是因为 `goalPointXY` 可能不会正好落在方格边线的交界处。在 8 个主要方向上邻近的节点都会被返回——只要 `SpaceTester` 显示出它们没有障碍物，或落在游戏世界的边界以内。

注意，邻近点的坐标值被四舍五入成 `stepSize` 的倍数，以防止发生聚集错误。如果只是简单地加上 `stepSize` 或减去 `stepSize`，那么有些值，例如 1.500001，就会被视为与 1.5 相同。这样就会导致某一个节点会被 A* 算法探测两次。

```

public double getCostToAdjacentNode (
    Object fromNode,
    Object toNode )
{
    ///////////////////////////////////////////////////
    return ( ( PointXY ) fromNode ).distanceXY ( ( PointXY ) toNode );
}

public boolean isGoalNode ( Object node )
{
    ///////////////////////////////////////////////////
    return goalPointXY.distanceXY ( ( PointXY ) node ) == 0.0;
}

```

这里使用的是一个距离测量值而不是一个相等性检查，主要是因为 `goalPointXY` 可能是一个不同类的 `goalPointXY`，而不是 `node` 参数的 `goalPointXY`——即使它们二者都实现了接口 `PointXY`。

8.7 GradientCartographer

以我的经验看来，`GradientCartographer` 就是 `GridCartographer` 的一个变体，这个变体似乎可以更快地找到目标。这意味着我们可以更好地进行路径发现，同时还可以保持较高的帧速率。代替使用一个空间上均匀分布的矩形网格，将邻近节点分布在环绕当前节点、带有可变半径的一个环上。由于起始节点和当前节点之间的距离是不断增大的，所以相邻节点之间的距离也是不断增加的。这样在紧贴起始节点的周围可以进行精细地分解，随着向外移动，分解的粒度也就逐渐变粗。这样的效果就是规划的路径在障碍物的周围将会仔细地选择它的路线，但是当移动到足够远以后，就会作较大的跳跃式前进，这样就可以更快地向目标前进。如图 8-2 所示。

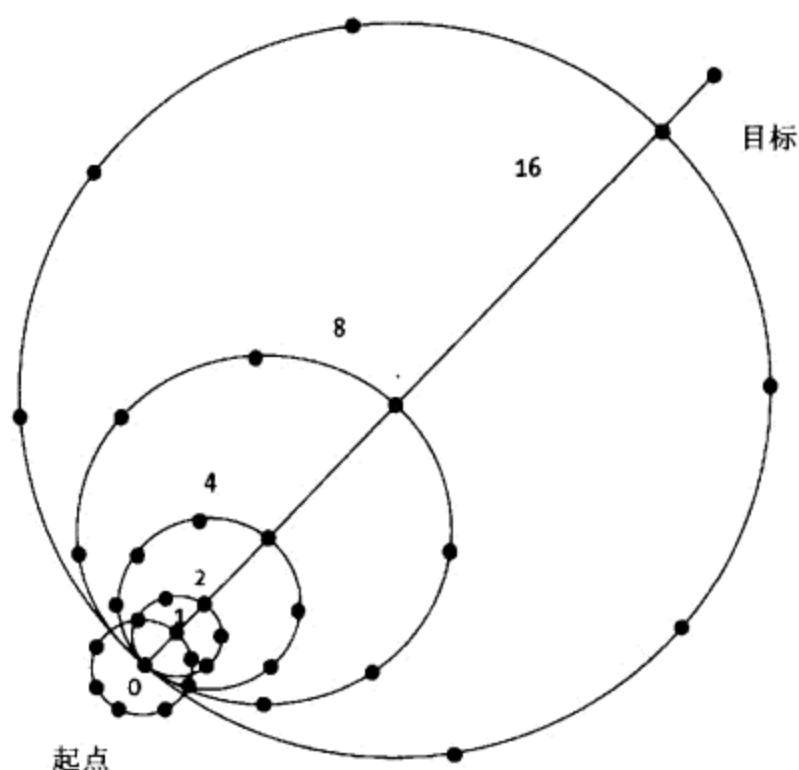


图 8-2 梯度节点空间

```

package com.croftsoft.core.ai.astar;

[...]

public final class GradientCartographer
    implements Cartographer
{
    //////////////////////////////////////
    //////////////////////////////////////
    {
        public static final double DEFAULT_INIT_STEP_SIZE = 1.0;

        public static final int DEFAULT_DIRECTIONS = 8;

        //

        private final List adjacentList;

        private final double initStepSize;

        private final SpaceTester spaceTester;

        private final int directions;

        //

        private PointXY startPointXY;

        private PointXY goalPointXY;
    }
}

```

`initStepSize` 是起始节点 `startPointXY` 和邻近节点的内环之间最初的半径。变量 `directions` 是环上节点的数目。如果希望能够搜索整个二维平面，最少需要在 3 个方向上进行搜索。默认的方向数是 8，这一点与 `GridCartographer` 中使用的方向数相同。

```
[...]

public Iterator getAdjacentNodes ( Object node )
///////////////////////////////////////////////////////////////////
{
    PointXY pointXY = ( PointXY ) node;

    adjacentList.clear ( );

    double distanceToGoal = pointXY.distanceXY ( goalPointXY );

    double distanceFromStart = pointXY.distanceXY ( startPointXY );

    double stepSize
        = ( ( int ) ( distanceFromStart / initStepSize ) ) * initStepSize;

    stepSize = Math.max ( stepSize, initStepSize );

    if ( distanceToGoal <= stepSize )
    {
        adjacentList.add ( goalPointXY );

        return adjacentList.iterator ( );
    }
}
```

stepSize 基本等于 distanceFromStart，四舍五入为 initStepSize 的倍数。再对它进行检查以保证它最小也为 initStepSize，以防止最初步的步长为 0。通过检查 distanceToGoal 是否小于或等于 stepSize，来防止超过目标。

```
double x = pointXY.getX ( );

double y = pointXY.getY ( );

double headingToGoal = Math.atan2 (
    goalPointXY.getY ( ) - y,
    goalPointXY.getX ( ) - x );

for ( int i = 0; i < directions; i++ )
{
    double heading = headingToGoal + i * 2.0 * Math.PI / directions;

    PointXY step = new Point2DD (
        x + stepSize * Math.cos ( heading ),
        y + stepSize * Math.sin ( heading ) );

    if ( spaceTester.isSpaceAvailable ( step ) )
    {
        adjacentList.add ( step );
    }
}

return adjacentList.iterator ( );
}
```


在此没有使用 8 个主要方向，而是将邻近节点的数目均匀地分布在以当前节点为中心的圆环上。第一个邻近节点的方向与目标节点的相同，以便降低路径中的变数。

8.8 TankConsole

```
package com.croftsoft.apps.mars.ai;

import java.awt.Shape;

import com.croftsoft.core.ai.astar.SpaceTester;
import com.croftsoft.core.math.geom.PointXY;

import com.croftsoft.apps.mars.model.TankAccessor;

public interface TankConsole
    extends TankAccessor, SpaceTester
    //////////////////////////////////////
    //////////////////////////////////////
    {

    public int getAmmo ( );

    public double getBodyHeading ( );

    public double getBodyRotationSpeed ( );

    public Shape getShape ( );

    public double getTankSpeed ( );

    public double getTurretHeading ( );

    //////////////////////////////////////
    //////////////////////////////////////

    public PointXY getClosestAmmoDumpCenter ( );

    public PointXY getClosestEnemyTankCenter ( );

    //

    public void fire ( );

    public void go ( PointXY destination );

    public void rotateTurret ( PointXY targetPointXY );
```

com.croftsoft.apps.mars.ai 包中游戏特定的类 TankConsole 提供了 Tank 实现的一个有限制的接口。在创建游戏特定的 AI 类时，我喜欢为模型类创建一个操作控制台接口，该接口只提供一部分有效的 accessor 方法和 mutator 方法。这种接口用来提醒什么是 AI 可以访问的信息，以及它们可以进行什么样的控制，也就是传感器和受动器。理想情况下，AI 驱动的敌方信息应

该仅限在受玩家自己支配的信息上。同样，控制的范围也应该仅限在受玩家自己支配的信息上——如果玩家不希望 AI 欺骗的话。

注意，TankConsole 实现了接口 SpaceTester，以便确定某一个具体的位置是空旷的还是被阻塞的。这类似于透过坦克窗口判断路径中有没有什么障碍物的模拟。AI 研究领域内的很多人都批评计算机模拟，因为它经常会带来一些捷径，而如果在机器人上实现 AI，就不能使用这些捷径。例如，机器人上的照相机返回的会是一组像素数据，而不是像 isSpaceAvailable() 这样的方法返回一个布尔值。机器人再通过 AI 技术来解释这个图像，以确定路径是否是畅通的。

我认为，机器人技术研究方面的计算机模拟将会有很好的发展前景。计算机模拟有这种优势：在同步运行在许多分布式计算机上的多个虚拟世界里，可以在加速的模拟时间内执行 AI 的训练和测试。但是一定要非常小心，以保证模拟中的操作控制台接口与在机器人上使用的操作控制台接口完全相同。理想的情况是，应该可以在一个虚拟的世界里训练 AI，再将它转移给一个机器人，而它感觉不到有什么不同——只要模拟的输入是真实的。

8.9 TankOperator

```
package com.croftsoft.apps.mars.ai;

import java.util.*;
import com.croftsoft.core.math.geom.PointXY;

public interface TankOperator
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    public void fire ( );

    public void go ( PointXY destination );

    public void setTankConsole ( TankConsole tankConsole );

    public void update ( double timeDelta );

    public Iterator getPath ( );
}
```

TankOperator 接口的实现通过 TankConsole 操作 Tank。只要在游戏循环内部调用了它的 update() 方法，它都会这样做。由于提供了 getPath() 方法，所以可以显示 TankOperator 已规划好的路径。

fire() 和 go() 命令由另一个实体(例如坦克指挥官或玩家)用来重写或补充 TankOperator 的 AI。这些命令可能会由 TankOperator 转发给 TankConsole。例如，一个坦克指挥官可以在一个指定的集合地点向 TankOperator 发送无线电信号。然后 TankOperator 可以使用它的 AI，在某个时候给 TankConsole 一个中间步骤，来指导坦克绕过障碍物到达最终的目标。

8.10 StateSpaceNode

```
package com.croftsoft.apps.mars.ai;

import com.croftsoft.core.math.geom.Point2DD;
import com.croftsoft.core.math.geom.PointXY;

public final class StateSpaceNode
///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////
{

    private final Point2DD point2DD;

    //

    private double heading;
```

在前面的 A*算法实现的示例中，我们使用的是二维整数或实数的位置点，作为状态空间节点的表示形式。但是在 Mars 游戏中，我们除了需要跟踪空间中的位置点以外，还要跟踪坦克的航向。这是因为坦克在移动之前，必须要调整前进的方向。在计算到达目标的路径时，必须要将这个旋转的时间考虑进去。

```
[...]

public double distance ( StateSpaceNode otherStateSpaceNode )
///////////////////////////////////////////////////////////////////
{
    return point2DD.distance ( otherStateSpaceNode.point2DD );
}

public double rotation ( StateSpaceNode otherStateSpaceNode )
///////////////////////////////////////////////////////////////////
{
    double otherHeading = otherStateSpaceNode.heading;

    double headingDelta = otherHeading - heading;

    if ( headingDelta < -Math.PI )
    {
        headingDelta = ( otherHeading + 2.0 * Math.PI ) - heading;
    }
    else if ( headingDelta > Math.PI )
    {
        headingDelta = ( otherHeading - 2.0 * Math.PI ) - heading;
    }

    return headingDelta;
}
```

Mars 游戏 AI 的 StateSpaceNode 提供了状态空间里的两个测量距离。一个是笛卡尔坐标系 (直角坐标系) 下虚拟世界里的常规距离。另一个是调整航向所需要的旋转角度。

8.11 TankCartographer

```
package com.croftsoft.apps.mars.ai;

[...]

public final class TankCartographer
    implements Cartographer
    //////////////////////////////////////
    //////////////////////////////////////
    {

    [...]

    public double getCostToAdjacentNode (
        Object fromNode,
        Object toNode )
    //////////////////////////////////////
    {
        StateSpaceNode fromStateSpaceNode = ( StateSpaceNode ) fromNode;

        StateSpaceNode toStateSpaceNode = ( StateSpaceNode ) toNode;

        double rotation = fromStateSpaceNode.rotation ( toStateSpaceNode );

        rotation = Math.abs ( rotation );

        double bodyRotationSpeed = tankConsole.getBodyRotationSpeed ( );

        double rotationTime = rotation / bodyRotationSpeed;

        double travelTime = calculateTravelTime ( fromNode, toNode );

        double totalTime = travelTime + rotationTime;

        return totalTime;
    }

    [...]

    private double calculateTravelTime (
        Object fromNode,
        Object toNode )
    //////////////////////////////////////
    {
        StateSpaceNode fromStateSpaceNode = ( StateSpaceNode ) fromNode;

        StateSpaceNode toStateSpaceNode = ( StateSpaceNode ) toNode;

        double distance = fromStateSpaceNode.distance ( toStateSpaceNode );
```



```

    return distance / tankConsole.getTankSpeed ( );
}

```

TankCartographer 查找邻近节点的技术与在 GradientCartographer 中使用的技术相同。但是从 `getCostToAdjacentNode()` 方法中返回的值不同，因为改变航向的时间也是一个要考虑的因素（不仅仅是移动的距离）。从一个状态空间节点到另一个状态空间节点的距离，是通过将距离转换成移动的时间，再加上旋转的时间来计算的。如果使用距离作为代价的计算单位，那么 A* 算法就会设计一个到达目标距离最小的路径。这就可能需要在这一路上让航向发生很多笨重且缓慢的变化。如果使用时间作为代价的计算单位，A* 算法就会设计一个最快到达目标的路径。这可能会增加移动的总距离，因为整个移动的路径将会被分成较长的一些折线段。

8.12 DefaultTankOperator

```

package com.croftsoft.apps.mars.ai;

[...]

public final class DefaultTankOperator
    implements TankOperator, Serializable
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

    /** Probability of firing during one second of time. */
    private static final double FIRING_PROBABILITY = 1.0;

    /** Probability of drifting during one second of time. */
    private static final double DRIFT_PROBABILITY = 0.1;

    private static final int A_STAR_LOOPS = 100;

    private static final double STEP_SIZE = 10.0;

    private static final int DIRECTIONS = 8;

```

DefaultTankOperator 是 TankOperator 的默认实现，这个默认实现控制与玩家对抗的敌方坦克。在单个更新过程中，它将 A* 循环的最大次数限制在 100 以内，这样计算所花费的时间就不会特别多。

```

[...]

public DefaultTankOperator ( Random random )
    //////////////////////////////////////
{
    NullPointerException.check ( this.random = random );

    center = new Point2DD ( );

```

```

destination = new Point2DD ( );

tankCartographer = new TankCartographer ( STEP_SIZE, DIRECTIONS );

aStar = new AStar ( tankCartographer );

startStateSpaceNode = new StateSpaceNode ( );
}

```

构造函数用 TankCartographer 初始化一个 Astar 实例。

[...]

```

public void update ( double timeDelta )
///////////////////////////////////////////////////
{
    ShapeLib.getCenter ( tankConsole.getShape ( ), center );

    enemyCenter = tankConsole.getClosestEnemyTankCenter ( );

    tankConsole.rotateTurret ( enemyCenter );
}

```

update()方法让坦克的枪口一直瞄准最近的敌人，而不管坦克体当前的航向。

```

if ( tankConsole.getAmmo ( ) < 1.0 )
{
    PointXY ammoDumpCenter
        = tankConsole.getClosestAmmoDumpCenter ( );
    if ( ammoDumpCenter != null )
    {
        tankConsole.go ( getFirstStep ( ammoDumpCenter ) );
    }

    return;
}

```

如果坦克弹药用完了，它就会寻找弹药库。它调用私有的 getFirstStep()方法判断怎样到达弹药库。

```

if ( enemyCenter != null )
{
    tankConsole.go ( getFirstStep ( enemyCenter ) );

    if ( random.nextDouble ( ) < timeDelta * FIRING_PROBABILITY )
    {
        tankConsole.fire ( );
    }

    return;
}

```

否则它就会向最近的敌人前进，边移动边开火。注意，开火的可能性，即 FIRING_PROBABILITY，是根据每秒积累的概率给出的。但是，timeDelta 总是会比一秒要小

很多，因为它是帧速率的倒数。简单地将每秒积累的概率与 `timeDelta` 的乘积作为调整后的概率是不太精确的。为了搞明白我说的意思，可以考虑一个开火概率为每秒 50% 的情况。如果 `timeDelta` 正好是 0.5 秒，那么这段时间调整过的概率将会是 25%。在两个 0.5 秒内，坦克至少一次都不开火的概率将会为 75% 的平方，也就是 56.25%。为了计算出对给定的 `timeDelta` 而言，调整过的开火概率应该是多少才能使开火或不开火的累积概率各为 50%，将需要使用一个公式，执行对数和逆对数函数调用计算。

如果每秒开火的概率不是 50% 而是比 50% 小很多的 10%，那么调整以后 0.5 秒的开火概率将是 5%。在两个 0.5 秒内，不开火的概率将会是 95% 的平方，也即 90.25%。那么在两个 0.5 秒内，至少开火一次的概率将会为 9.75%，这就非常接近于 10%。如果再下调累积开火概率，例如将其调整到 1%，那么在两个 0.5 秒内至少开火一次的概率就会是 0.9975%，这与 1% 仅相差 0.0025%。由于在调整的概率中，累积概率越小，相对误差也就越小，所以您在这里可能只想使用简单的乘法，而不是更加复杂的数学公式。

```
if ( random.nextDouble ( ) < timeDelta * DRIFT_PROBABILITY )
{
    destination.setXY (
        center.x + 2 * random.nextDouble ( ) - 1,
        center.y + 2 * random.nextDouble ( ) - 1 );

    tankConsole.go ( destination );
}

if ( random.nextDouble ( ) < timeDelta * FIRING_PROBABILITY )
{
    tankConsole.fire ( );
}
}
```

如果坦克还有弹药，但没有找到要进行攻击的敌人坦克，那么它就会随机地在这个区域进行巡逻，并会盲目开火(特殊情况下)。

[...]

```
private PointXY getFirstStep ( PointXY destination )
///////////////////////////////////////////////////
{
    NullPointerException.check ( destination );

    startStateSpaceNode.setPointXY ( center );

    startStateSpaceNode.setHeading ( tankConsole.getBodyHeading ( ) );

    tankCartographer.setStartStateSpaceNode ( startStateSpaceNode );

    aStar.reset ( startStateSpaceNode );

    tankCartographer.setGoalPointXY ( destination );
}
```

私有方法 `getFirstStep()` 首先将 `startSpaceNode` 初始化为当前坦克所在的位置和坦克的航向。然后它重设 A* 算法搜索，并通知 `tankCartographer` 将当前 `destination` 作为目标使用。如果虚拟游戏世界里的障碍物不是不断地四处移动，并且起始点和目标都没有发生变化，从一次更新到另一次更新中，就不需要重设搜索。当希望在多重更新阶段上延伸对目标的搜索，以便给 AI 更多思考时间的时候，这会很有好处。

```
for ( int i = 0; i < A_STAR_LOOPS; i++ )
{
    if ( !aStar.loop ( ) )
    {
        break;
    }
}
```

A* 搜索将会搜索目标的整个 `A_STAR_LOOPS` 状态空间节点。无论 `A_STAR_LOOPS` 是多大，它永远也不能够作一个完全的穷尽搜索，因为游戏世界的大小是无限的。与接口 `Cartographer` 的 `AstarTest` 实现不同，`TankCartographer` 并没有将邻近的节点仅限制在某一个设定区域内的那些节点上。如果目标完全被障碍物阻挡，这样从起始点就没有路径可以与之相通，那么只有在满足某种限制条件后循环才能退出。

```
if ( !aStar.isGoalFound ( ) )
{
    return destination;
}
```

如果没有找到目标，就会将最后的 `destination` 作为第一步返回。这意味着，如果在分配的计算时间内，不能找到一个通往目标的完全路径，那么这个坦克就会简单地直接向目标移动一段距离。这也是合理的策略，因为这通常都是最好的移动。坦克一旦靠近目标，它就可以找到一个通往目标的路径了。

```
StateSpaceNode stateSpaceNode
    = ( StateSpaceNode ) aStar.getFirstStep ( );

if ( stateSpaceNode == null )
{
    return destination;
}

return stateSpaceNode.getPointXY ( );
}
```

如果找到了目标，就会向 `Astar` 实例查询路径的第一步。我们知道，即使找到了目标，但是如果起始点和目标点在同一个位置上，`Astar` 的实例也可能会返回 `null`。这是因为路径的第一步通常会跳过起始点。

8.13 PlayerTankOperator

PlayerTankOperator 通过 TankConsole 接口控制玩家坦克。

```
package com.croftsoft.apps.mars.ai;

import java.util.*;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.math.geom.Point2DD;
import com.croftsoft.core.math.geom.PointXY;
import com.croftsoft.core.math.geom.ShapeLib;
import com.croftsoft.core.util.NullIterator;

public final class PlayerTankOperator
    implements TankOperator
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

    /** milliseconds */
    private static final long AUTO_PILOT_DELAY = 15 * 1000;

    //
    private final TankOperator autoPilotTankOperator;
```

autoPilotTankOperator 将在玩家停止操作 15 秒以后获得控制权。它只是指向 DefaultTankOperator 实例的一个接口引用。

```
private final List pathList;

private final StateSpaceNode stateSpaceNode;
```

TankOperator 接口提供了一个 getPath() 方法，该方法的用途是，为在屏幕上的虚拟投影返回一个包含规划路径中 StateSpaceNode 实例的一个 Iterator。当玩家通过鼠标向 TankController 输入命令，手动控制坦克时，pathList 里就只有一个 stateSpaceNode。

```
private TankConsole tankConsole;

private boolean      autoPilotMode;

private boolean      fireRequested;

private PointXY      destination;

private boolean      destinationRequested;

private long         lastInputTime;

/////////////////////////////////////////////////
```

```

////////////////////////////////////
public PlayerTankOperator ( TankOperator autoPilotTankOperator )
////////////////////////////////////
{
    NullPointerException.check (
        this.autoPilotTankOperator = autoPilotTankOperator );

    pathList = new ArrayList ( 1 );

    stateSpaceNode = new StateSpaceNode ( );

    pathList.add ( stateSpaceNode );

    lastInputTime = System.currentTimeMillis ( );
}

////////////////////////////////////
////////////////////////////////////

public void fire ( )
////////////////////////////////////
{
    fireRequested = true;
}

public void go ( PointXY destination )
////////////////////////////////////
{
    destinationRequested = true;

    this.destination = destination;

    stateSpaceNode.setPointXY ( destination );
}

```

当 TankController 接收到鼠标和键盘的输入以后，它就会调用 PlayerTankOperator 的 fire() 和 go() 方法。PlayerTankOperator 并不会立刻处理这些请求，而是等到调用 update() 方法时才处理这些请求。这样就只允许在游戏循环的更新阶段才可以更新 Tank 模型的状态，在此外的任何时候都不能进行更新。

```

public Iterator getPath ( )
////////////////////////////////////
{
    if ( autoPilotMode )
    {
        return autoPilotTankOperator.getPath ( );
    }

    if ( destination == null )
    {

```



```

        return NullIterator.INSTANCE;
    }

    return pathList.iterator ( );
}

[...]

public void update ( double timeDelta )
////////////////////////////////////
{
    if ( fireRequested || destinationRequested )
    {
        autoPilotMode = false;

        lastInputTime = System.currentTimeMillis ( );

        if ( fireRequested )
        {
            tankConsole.fire ( );
            fireRequested = false;
        }

        if ( destinationRequested )
        {
            tankConsole.go ( destination );

            tankConsole.rotateTurret ( destination );

            destinationRequested = false;
        }
    }
    else if ( autoPilotMode )
    {
        autoPilotTankOperator.update ( timeDelta );
    }
    else if (
        System.currentTimeMillis ( ) > lastInputTime + AUTO_PILOT_DELAY )
    {
        autoPilotMode = true;
    }
}

```

如果 PlayerTankOperator 还处在 autoPilotMode 模式的时候, 用户移动了鼠标或按下了键盘, 那么就会恢复手动控制, 直到用户再停止 15 秒以后, 才再次进入到自动模式。

8.14 小结

在游戏程序设计中, A*算法的主要用途是查找绕过障碍物到达目标的路径, 但是这个算法也可以用在所有需要进行启发式状态空间搜索的领域。本章详细介绍了 A*算法的一个 Java 实

现。通过接口 Cartographer 的一个实现，将游戏特定的逻辑和游戏世界的数据分隔开，提高了 A* 算法的这个 Java 实现的可重用性。分配给每一个动画帧进行状态空间搜索的总处理时间，可以通过限制对 Astar 的 loop() 方法调用的次数来进行限制。两个动态产生的邻近节点之间的梯度间隔可以用来减少搜索时间。本章使用 Mars 游戏中路径投影选项，对在有移动障碍物的 2D 连续空间中的 A* 路径规划进行了可视化的演示。

8.15 参考文献

Bigus, Joseph P. and Jennifer Bigus. *Constructing Intelligent Agents Using Java*, 2nd edition. Hoboken, NJ: John Wiley & Sons, 2001.

DeLoura, Mark (ed.). "The Basics of A* for Path Planning." Chapter 3.3 in *Game Programming Gems*. Hingham, MA: Charles River Media, 2000.

Laird, John E. and Michael van Lent. "Human-Level AI's Killer Application: Interactive Computer Games." *AI Magazine*, Summer 2001.

Luger, George F. and William A. Stubblefield. "Heuristic Search." Chapter 4 in *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 2nd edition. San Francisco, CA: Benjamin/Cummings Publishing Co., 1993.

Watson, Mark. *Intelligent Java Applications for the Internet and Intranets*. San Francisco, CA: Morgan Kaufmann Publishers, 1997.



第 9 章 HTTP 隧道技术

*那些可以放弃基本的自由去获得一点点暂时安全的人们，
既不会得到自由也不会得到安全。*

——本杰明·富兰克林

网络安全防火墙为网络游戏的编程人员带来了特殊的挑战。防火墙可能会阻断除了往外发送的 Web 浏览器请求和它们的响应以外的所有网络连接。由于无论在家里还是在办公室里，防火墙都越来越普遍，所以游戏客户端的网络连接代码需要能够自动穿越这些防火墙，将消息传送到游戏服务器，而不需要玩家或网络管理员进行干预。

超文本传输协议(Hypertext Transfer Protocol, HTTP)隧道就常常用来完成这种任务。一般都允许 HTTP 穿越防火墙，因为 HTTP 本身就是 Web 浏览器使用的协议。为了向外发送一个消息，游戏客户端会把它的消息以 HTTP 请求的形式投递给服务器，就像它是从 Web 服务器上请求一个 Web 页面一样。但是，游戏服务器不是返回一个 HTML 格式的 Web 页面，而是返回一个经过编码的响应，客户端可以解析这种响应。

警告：

在 1997 年末的一个工作面试中，我试图演示一个动画的聊天服务器，这个聊天服务器是我使用简单的套接字连接而不是使用 HTTP 连接实现的。让我十分困惑的是，这个演示程序完全失败了，因为客户端 applet 一直不能和服务器建立连接。由于这个应用程序自始至终都是在我家里开发和测试的，所以我完全忘记了办公环境中常用的安全限制。结果您可能已经预料到了，老板不愿意在他们的防火墙上为我专门开一个口子，也正是因为这样，我也才得以完成我的这个示例程序。

Java 开发人员可以使用的网络协议和 API 有很多。其中包括同步远程过程调用(remote procedure call, RPC)技术，例如 RMI、JDBC socket 和简单对象访问协议(Simple Object Access Protocol, SOAP)。异步通信技术包括 MOM(面向消息的中间件, message-oriented middleware)的 JMS API、企业之间可靠通信的 JAXM (Java Architecture for XML Messaging)和 P2P(点对点, peer-to-peer)的 Juxtapose(JXTA)体系。最终，所有的这些技术都要采取 HTTP 隧道，才能让它们的消息越过网络防火墙。

但是，在用于 Java 游戏编程方面的时候，上述这些协议中的每一个都有一定的缺点。SOAP、JMS、JAXM 和 JXTA 需要像 J2EE 这样的 API 或库，它们在客户机上通常都不是预安装的。RMI 是包含在核心 J2SE 中的，但是它需要 ORB(对象请求代理, object request broker)的服务器端设置，并限制在客户机和服务器上都必须使用 Java 语言。

另一方面，HTTP 网络在客户机上不需要任何比核心 J2SE 还高级的东西，只需要几个自定义的类。这使它对小型的基于 applet 的游戏而言非常适用。在服务器上，一个简单的 servlet

容器¹就足够了。很多回避了 RMI ORB 安装的商业 Web 主机服务,或共享 Web 服务器上的一个定制游戏服务器都可以毫无问题地提供 servlet 支持。为了使游戏服务器与非 Java 客户软件更加兼容,将消息编码从串行化的 Java 对象改为纯文本格式(例如 XML)也相当容易。

本章主要介绍基于 HTTP 的客户机-servlet 通信所采用的一个灵巧的联网库。首先通过介绍一个示例应用程序,详细介绍这个库类和它们的使用方法。然后详细介绍了客户机和服务器的源代码,包括与可重用网络库类有一定区别的、游戏特定的应用程序代码。游戏重用这个库的时候,应该设法使在几乎所有的开发环境中都可以运行这些网络功能。

9.1 测试示例

这一章里的示例代码依赖于 Servlet API 框架。虽然客户机上只需要 J2SE,但是在服务器上,还需要像 Tomcat 或 Jetty²这样的一个 servlet 容器。这些 servlet 容器曾经都被称为“应用程序服务器”。但是现在“应用程序服务器”这个术语通常专指那些提供额外的 J2EE API(例如 EJB)的服务器,并且现在应用程序服务器还可能内嵌一个 servlet 容器。例如,开放源代码应用程序服务器 JBoss,就可以和已经与它集成的 Jetty 或 Tomcat 一起进行下载³。

最近,我曾用 Tomcat 开发全部的 servlet,因为它是 Servlet API 引用实现。现在我更加喜欢使用 JBoss,这主要有两个原因。第一个原因是 JBoss 是一个强大的应用程序服务器,在开发环境和部署环境中都可以免费使用。第二个原因是 JBoss 支持 WAR 文件的自动部署。当 Ant 构建将更新后的 WAR 文件复制到 JBoss 部署目录上时,JBoss 就自动检测这种变化,并升级正在运行游戏的 servlet,而不需要先关闭游戏 servlet,最后再重新启动它。这在开发过程中非常有用。如果您还没有使用 JBoss,我推荐您应该立即下载并安装它。

注意:

您还应该知道,最近媒体宣布了一个新的 Apache 项目,叫作 Geronimo⁴。如果成功的话,它可能会是 JBoss 的另一种选择。那时您将能够在两个免费的开发源代码的 J2EE 容器之间作出选择了。

还需要将 J2EE SDK 下载到您的开发机器上,这样,就可以独立于 servlet 容器库来编译代码了⁵。在 CroftSoft 代码库的构建文件 build.xml 中,需要将属性 J2EE_HOME 更改为指向 J2EE SDK 的安装目录。这样做之后,就应该能够成功地使用 Ant 的目标 score 编译和打包这个示例 servlet。

假设没有出现编译错误,接着就需要将属性 deploy_dir 更改为指向 servlet 容器 WAR 部署目录。可能需要使用编辑器的搜索功能来查找这个属性是在这个构建文件的哪个位置定义的。当运行 Ant 目标 score_install 时,它就会首先运行目标 score,然后再将新的 WAR 文件复制到 deploy_dir。

```
appletviewer http://localhost:8080/score/
```

- 1 <http://java.sun.com/products/servlet/>
- 2 <http://jakarta.apache.org/tomcat/>, <http://jetty.mortbay.org/>
- 3 <http://www.jboss.org/>
- 4 <http://incubator.apache.org/projects/geronimo.htm>
- 5 <http://java.sun.com/j2ee/>

为了避免与另一个可能正在运行的 Web 服务器发生冲突,也为了避免与第一个 1024 端口号上强加的某些安全限制发生冲突, Tomcat 和 Jboss 最初都被配置为使用 HTTP 的 8080 端口,而不是 HTTP 默认的 80 端口。假设没有重新配置 HTTP 端口号,并且正在与服务器相同的机器 localhost 上运行客户程序,就可以使用上面给出的 `appletviewer` 命令来启动和测试这个示例代码。我发现在开发过程中,使用 `appletviewer` 进行 applet 测试比使用 Web 浏览器进行 applet 测试要更加方便。

在前面,我曾提到了使用服务器端代码,可以在无符号 applet 中保存持久数据,例如最高分(如图 9-1 所示)。这个示例代码介绍了使用 HTTP 隧道的能力。当首次初始化 applet 的时候,它会建立与对它提供下载功能的服务器的一个网络连接,并从持久存储器中,检索以前的最高得分。这个示例“游戏”的“玩法”是通过观察当前分数从 0 开始自动递增进行的。可以在任何时候单击屏幕结束游戏。如果在那一点的得分超过了从服务器检索到的最高分,那么服务器就会被更新,游戏就会在重新开始。然后,可以确认代码是按上传的方式进行工作的,方法是将 applet 和服务端都关闭,然后再将二者重新启动,并注意观察最新的最高分已被成功调出。



图 9-1 示例 HTTP 客户端

9.2 可重用的客户端代码

作为示例 applet 的一部分,本节首先介绍在网络连接的客户机上使用的多个可重用接口和类。虽然下面会详细介绍每一个类,但是在这里还是首先要对它们进行一下简要说明。Encoder 负责将外发消息对象转换为一个字节数组。Parser 负责从 `InputStream` 中解析接收到的信息对象。`StreamLib` 提供从 `InputStream` 中读取 `String` 的一个静态方法。`StringCoder` 用作操作纯文本格式消息对象的一个 Encoder 和 Parser。`HttpLib` 提供了一个向服务器发送消息并返回解析后响应的一个静态方法。`Queue` 负责保存发送和接收到的信息。`ListQueue` 是一个由 `List` 的一个实现(例如 `LinkedList`)支持的 `Queue`。重复操作都封装在 `Loopable` 中。`Looper` 在一个线程内部循

环这个 Loopable，这个线程是由 Lifecycle 方法进行管理的。HttpMessagePusher 使用 Looper 管理将一个消息从发送队列中拖出并将它发送给服务器的这种重复操作。

9.2.1 Encoder

包 com.croftsoft.core.io 中的接口 Encoder 实现用于将内存中对象转换为一个字节数组，以便于通过网络进行传输。专门使用一个接口的好处是，带来了交换编码实现的灵活性。例如，为了将需要传输的字节数量降到最小，客户端可能会以压缩的串行化对象的形式发送请求信息。为了与用其他编程语言编写的服务器程序进行通信，我们可能会选用 XML。

```
package com.croftsoft.core.io;

import java.io.*;

public interface Encoder
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

    public byte [ ] encode ( Object object )
        throws IOException;
```

9.2.2 Parser

接口 Parser 执行的操作与 Encoder 执行的操作相反。客户端程序使用 Parser 实现解析从服务器传来的响应，将字节流转换为一个对象。服务器程序也可以使用这种解析器解析来自于客户端的请求。

```
package com.croftsoft.core.io;

import java.io.*;

public interface Parser
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

    public Object parse (
        InputStream inputStream,
        int          contentLength )
        throws IOException;
```

contentLength 值 - 1 表示这个 contentLength 是未知的。

9.2.3 StreamLib

类 StreamLib 中的静态方法 toString() 使用指定的字符集编码，将字节流转换为 String。常见的字符集编码包括 US-ASCII 和 UTF-8。这种编码并不是和 Java 内核类(例如 BufferedReader 和 StringWriter)一起指定的，因为它们是操作在字符流上而不是字节流上的。


```

package com.croftsoft.core.io;

[...]

public static String toString (
    InputStream inputStream,
    String encoding )
    throws IOException, UnsupportedEncodingException
    //////////////////////////////////////
{
    ByteArrayOutputStream byteArrayOutputStream
        = new ByteArrayOutputStream ( );

    int i;

    while ( ( i = inputStream.read ( ) ) > -1 )
    {
        byteArrayOutputStream.write ( i );
    }

    return byteArrayOutputStream.toString ( encoding );
}

```

9.2.4 StringCoder

StringCoder 是 Encoder 和 Parser 的一个实现，它使用一个指定的字符集编码将 String 对象转换为字节序列，或将字节序列转换为 String 对象。对发送和接收纯文本格式的消息来说，StringCoder 非常有用。

```

package com.croftsoft.core.io;

import java.io.*;

import com.croftsoft.core.lang.NullArgumentException;

public final class StringCoder
    implements Encoder, Parser
    //////////////////////////////////////
    //////////////////////////////////////
{

    public static final String US_ASCII = "US-ASCII";

    public static final String UTF_8 = "UTF-8";

    //

    private final String charSetName;

    //////////////////////////////////////
    //////////////////////////////////////

```

```

public StringCoder ( String charSetName )
///////////////////////////////////////////////////
{
    NullArgumentException.check ( this.charSetName = charSetName );
}

///////////////////////////////////////////////////
///////////////////////////////////////////////////

public byte [ ] encode ( Object object )
    throws IOException
///////////////////////////////////////////////////
{
    return object.toString ( ).getBytes ( charSetName );
}

public Object parse (
    InputStream inputStream,
    int          contentLength )
    throws IOException
///////////////////////////////////////////////////
{
    return StreamLib.toString ( inputStream, charSetName );
}

```

9.2.5 HttpLib

`com.croftsoft.core.net.http` 包中的静态方法库 `HttpLib` 提供了一个 `post()` 方法, 这个方法的使用是使用 HTTP 向服务器发送一个消息, 并解析服务器对消息作出的响应。

```

package com.croftsoft.core.net.http;

[...]

import com.croftsoft.core.io.Parser;
import com.croftsoft.core.io.StringCoder;

public final class HttpLib
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

    [...]

    public static Object post (
        URL          url,
        byte [ ] bytes,
        String      userAgent,
        String      contentType,
        Parser      parser )
        throws IOException
///////////////////////////////////////////////////
{

```


url 是要进行连接的服务器地址。参数 bytes 是要上传到服务器经过编码的消息。userAgent 向服务器标识客户软件及其版本号，例如 MyGameClient/1.0。contentType 告诉服务器该怎样解析请求，例如 text/xml。parser 用来解析并返回来自于服务器的响应——如果有的话。如果想忽略所有响应，可以给 parser 提供一个空值。

```

HttpURLConnection httpURLConnection
    = ( HttpURLConnection ) url.openConnection ( );

httpURLConnection.setRequestMethod ( "POST" );

if ( userAgent != null )
{
    httpURLConnection.setRequestProperty ( "User-Agent", userAgent );
}

if ( contentType != null )
{
    httpURLConnection.setRequestProperty (
        "Content-Type", contentType );
}

httpURLConnection.setRequestProperty (
    "Content-Length", Integer.toString ( bytes.length ) );

httpURLConnection.setDoOutput ( true );

OutputStream outputStream = httpURLConnection.getOutputStream ( );

BufferedOutputStream bufferedOutputStream
    = new BufferedOutputStream ( outputStream );

bufferedOutputStream.write ( bytes );

bufferedOutputStream.close ( );

if ( parser != null )
{
    if ( httpURLConnection.getResponseCode ( )
        == HttpURLConnection.HTTP_OK )
    {
        int contentLength = httpURLConnection.getContentLength ( );

        InputStream inputStream = httpURLConnection.getInputStream ( );

        BufferedInputStream bufferedInputStream
            = new BufferedInputStream ( inputStream );

        Object object
            = parser.parse ( bufferedInputStream, contentLength );

        bufferedInputStream.close ( );
    }
}

```

```

        return object;
    }
}

return null;
}

```

将字节流写到 `outputStream` 之后, 就会使用 `getResponseCode()` 方法检查是否有来自于服务器的响应。各种不同的 HTTP 响应代码是在内核类 `URLConnection` 中以整常量的形式定义的。浏览 Web 时最常返回的代码就是 `HTTP_OK`, 并带有一个值 200, 这表达了两层含义, 一层含义是请求已被处理, 并且没有产生错误, 另一层含义就是服务器给出了一个响应。如果没有来自于服务器的响应, 或者服务器返回了一个错误代码, 或者是没有 `parser` 来解析这个响应, `post()` 方法就返回空值。

```

System.out.println (
    post (
        new URL ( "http://localhost:8080/score/servlet" ),
        "get".getBytes ( StringCoder.UTF_8 ),
        "HttpLib/1.0",
        "text/plain",
        new StringCoder ( StringCoder.UTF_8 ) ) );

```

在使用了 `HttpLib post()` 方法的这个示例中, 使用 UTF-8 字符集编码将纯文本消息 "get" 发送给 `servlet`。进行发送的客户软件向服务器将自己的版本标识为 `HttpLib 1.0`。任何响应都会被解析并显示到标准的输出上。

这个 `post()` 方法给出了实现 HTTP 隧道的最低需求。但是, 使用与驱动动画相同的一个线程, 直接从游戏代码中调用这个方法, 可能会导致用户界面停止。如果网络不仅慢而且实际上性能还在不断下降, 那么用户界面可能就会被完全锁死。如果是这样, 那么直到网络连通性恢复以前, 结束应用程序进程或重新启动计算机可能是玩家的惟一选择。

9.2.6 Queue

在网络调用过程中, 为了避免阻塞主动画和游戏循环, 可以将请求消息追加到队列中。这使主线程(通常就是事件分派线程)可以立即进行处理, 而无需等待。由一个独立的线程连续地将这些请求拖出队列, 并通过网络逐个将它们上传。

```

package com.croftsoft.core.util.queue;

public interface Queue
////////////////////////////////////
////////////////////////////////////
{

    public boolean append ( Object o );

    public Object poll ( );

```



```

public Object pull ( )
    throws InterruptedException;

public Object pull ( long timeout )
    throws InterruptedException;

public Object replace ( Object o )
    throws IndexOutOfBoundsException;

```

`com.croftsoft.core.util.queue` 包中的接口 `Queue` 是队列实现的一个通用接口。方法 `append()` 用来向队列添加一个对象，它总会立即返回。如果队列为空，它可能就返回一个 `false`，表示追加操作没有成功。相应的方法 `pull()` 从队列中获取下一个对象。如果队列为空，`pull()` 方法将会将调用线程阻塞，直到有一个对象被添加到队列中。另一方面，方法 `poll()` 总会立即返回，即使队列中没有什么东西需要返回也是如此。

调用 `replace()` 的时候，它就会检查要添加到队列中的对象是否与队列中已经存在的某一个对象相等，这是使用 `Object` 类的方法 `equals()` 来进行一个相等性比较而得出的。如果有一个对象与要添加到队列中的这个对象相等，那么这个对象就会在队列中相同的位置上被新对象所替换。如果没有一个对象与要添加到队列中的这个对象相等，那么 `replace()` 方法就会简单地追加这个对象。当产生请求的速度比传输请求的速度更快，并且想保证在任何时候队列中都没有某种请求的多个实例时，这种处理就非常有用。

9.2.7 ListQueue

接口 `Queue` 的具体实现有两个：`ListQueue` 实现和 `VectorQueue` 实现。`ListQueue` 实现由一个带 `LinkedList` 的 `List` 支持，这个实现也是一个默认的实现。在 `ListQueue` 上调用 `append()` 方法实际上会委托给 `List.add()`。`Queue.poll()` 实现依赖于 `List.remove()`。使用 `ListQueue` 与 `List` 存储对象的主要不同之处在于，当 `Queue` 为空的时候，`ListQueue` 提供了阻塞 `pull()` 行为的线程。

提示：

过去，我用类 `VectorQueue` 而不是 `ListQueue` 来作为默认的实现，用以存储进出的网络消息，因为 `Vector` 与 Java 1.1 相兼容。那时，我正在致力于开发一个项目，在这个项目中，我在客户端上使用的是 Java 1.1，尽管服务器上已经安装了 Java 1.4，因为我们不想要求玩家升级它们的浏览器的 JVM。即使可能不需要 Java 插件程序(Plug-in)技术的这种先进性，但是如果确实要采取这种方法，在编译客户端代码的时候，请一定要记住，务必在 `javac` 命令中使用 `-target 1.1` 参数。

```

package com.croftsoft.core.util.queue;

import java.io.Serializable;
import java.util.*;

import com.croftsoft.core.lang.NullArgumentException;

public final class ListQueue
    implements Queue, Serializable
    //////////////////////////////////////

```

```

////////////////////////////////////
{

private static final long serialVersionUID = 0L;

//

private final List list;

private final int maxSize;

////////////////////////////////////
////////////////////////////////////

public ListQueue (
    List list,
    int  maxSize )
////////////////////////////////////
{
    NullPointerException.check ( this.list = list );

    this.maxSize = maxSize;
}

[...]

public boolean append ( Object o )
////////////////////////////////////
{
    NullPointerException.check ( o );

    synchronized ( list )
    {
        if ( list.size ( ) < maxSize )
        {
            list.add ( o );

            list.notifyAll ( );

            return true;
        }
    }

    return false;
}

```

append()方法会通知所有在等待的线程，有一个对象已经被添加到队列中去了。

```

public Object poll ( )
////////////////////////////////////
{
    synchronized ( list )

```



```

{
    if ( list.size ( ) > 0 )
    {
        return list.remove ( 0 );
    }
}

return null;
}

```

`poll()`方法在 `list` 上进行同步，这样就没有任何元素可以被另一个线程并发地移除。这保证了在索引位置 0 上的第一个元素可以被移除。

```

public Object pull ( )
    throws InterruptedException
    //////////////////////////////////////
{
    return pull ( 0 );
}

```

上面给出的这个不带参数的 `pull()`方法委托给了下面这个 `pull()`方法，而后者带有一个 `timeout` 参数。`timeout` 参数为 0 表示没有超时限制；调用线程将会一直被阻塞，直到被中断或有一个对象被添加到队列中。

```

public Object pull ( long timeout )
    throws InterruptedException
    //////////////////////////////////////
{
    if ( timeout < 0 )
    {
        throw new IllegalArgumentException ( "timeout < 0" );
    }

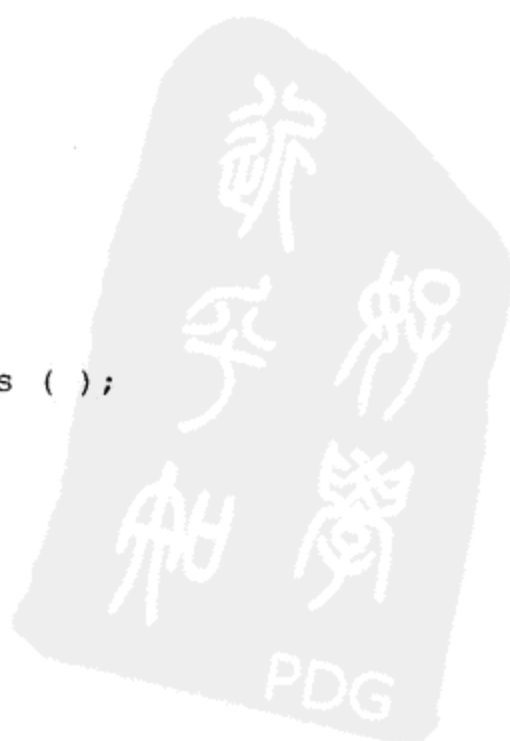
    long stopTime = System.currentTimeMillis ( ) + timeout;

    Object o = null;

    while ( ( o = poll ( ) ) == null )
    {
        if ( timeout == 0 )
        {
            synchronized ( list )
            {
                list.wait ( );
            }
        }
        else
        {
            long nowTime = System.currentTimeMillis ( );

            if ( stopTime > nowTime )
            {
                synchronized ( list )

```



```

        {
            list.wait ( stopTime - nowTime );
        }
    }
    else
    {
        break;
    }
}

return o;
}

```

如果首次调用 `pull(timeout)` 方法的时候，队列中没有任何对象，`pull(timeout)` 方法就会一直等待，直到它被通知有一个对象已经添加到队列中，线程被中断或者已过 `timeout` 期限。

```

public Object replace ( Object o )
    throws IndexOutOfBoundsException
    ////////////////////////////////////////
{
    synchronized ( list )
    {
        int index = list.indexOf ( o );

        if ( index < 0 )
        {
            if ( append ( o ) )
            {
                return null;
            }

            throw new IndexOutOfBoundsException ( );
        }
        else
        {
            return list.set ( index, o );
        }
    }
}

```

`VectorQueue` 的 `replace()` 方法使用 `indexOf()` 方法判定队列中是否存在一个与它相等的对象。`indexOf()` 方法对队列中的对象调用 `equals()` 作这种比较。`Object` 类中 `equals()` 方法默认的实现是执行一个简单的标识比较，也就是比较操作数是否指向内存中同一个对象。很多类(例如 `String` 和 `Long`)都重写了默认的 `equals()` 实现，以便于通过值而不是标识进行比较。

9.2.8 Loopable

发送队列和接收队列都将使用带 `pull()` 方法的线程处理消息。这些线程会连续循环，将消息对象从队列中拖出并依次处理。


```
package com.croftsoft.core.util.loop;

public interface Loopable
////////////////////////////////////
////////////////////////////////////
{

    public boolean loop ( );
}
```

`com.croftsoft.core.util.loop` 包中的接口 `Loopable` 为单个循环操作提供了一个通用的方法签名。如果循环应该继续运行，方法 `loop()` 就会返回 `true`，否则就返回 `false`。

9.2.9 Looper

理想的情况下，我们希望能够在游戏暂停的时候停止网络通信。这需要使用生命周期方法管理运行接收和发送网络消息队列循环的那些线程。类 `Looper` 是为了进行即通用又可重用的循环管理而实现了接口 `Lifecycle` 的一个类。

```
package com.croftsoft.core.util.loop;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.ex.ExceptionHandler;
import com.croftsoft.core.lang.lifecycle.Lifecycle;

public final class Looper
    implements Lifecycle
    //////////////////////////////////
    //////////////////////////////////
{

    private final Loopable      loopable;

    private final ExceptionHandler exceptionHandler;

    private final String        threadName;

    private final int            threadPriority;

    private final boolean       useDaemonThread;

    //

    private LoopGovernor loopGovernor;
    private Thread        thread;
    private boolean        stopRequested;
```

`Looper` 将会周期性地执行它的 `loopable`。如果 `loopable` 被损坏，就会产生一个 `exceptionHandler` 决定怎样处理这个错误。运行这个循环的线程是由 `threadName` 进行标识的，并且这个线程以由给定的 `threadPriority` 确定的优先级运行。如果设置了 `useDaemonThread`，在关闭游戏的时候，循环就会自动终止。`loopGovernor` 管理 `thread` 循环的速度。可以通过设置 `stopRequested` 将这个循环临时挂起。

```

public Looper (
    Loopable          loopable,
    LoopGovernor      loopGovernor,
    ExceptionHandler  exceptionHandler,
    String            threadName,
    int                threadPriority,
    boolean            useDaemonThread )
    //////////////////////////////////////
{
    NullPointerException.check ( this.loopable = loopable );

    setLoopGovernor ( loopGovernor );

    this.exceptionHandler = exceptionHandler;

    this.threadName       = threadName;

    this.threadPriority    = threadPriority;

    this.useDaemonThread  = useDaemonThread;
}

public Looper ( Loopable loopable )
    //////////////////////////////////////
{
    this (
        loopable,
        new FixedDelayLoopGovernor ( 0L, 0 ),
        ( ExceptionHandler ) null,
        ( String ) null,
        Thread.MIN_PRIORITY,
        true );
}

```

这个便利构造函数为主构造函数的参数提供了多个默认值。默认的 LoopGovernor 实现是带有 0 延迟的一个 FixedDelayLoopGovernor。0 延迟将会让线程休眠的长度正好可以检查是否还有其他线程需要处理时间。

[...]

```

private void loop ( )
    //////////////////////////////////////
{
    while ( thread != null )
    {
        try
        {
            if ( loopable.loop ( ) )
            {
                loopGovernor.govern ( );
            }
        }
        else
    }
}

```



```

        {
            stopRequested = true;
        }
    }
    catch ( InterruptedException ex )
    {
    }
    catch ( Exception ex )
    {
        if ( ( exceptionHandler == null )
            || !exceptionHandler.handleException ( ex, loopable ) )
        {
            stopRequested = true;

            ex.printStackTrace ( );
        }
    }

    if ( stopRequested )
    {
        synchronized ( this )
        {
            while ( stopRequested )
            {
                try
                {
                    wait ( );
                }
                catch ( InterruptedException ex )
                {
                }
            }
        }
    }
}

```

Looper 中的生命周期方法 `init()`、`start()`、`stop()`和 `destroy()`的实现几乎与前面介绍的 Swing 动画中这些方法完全相同。主要的区别在于私有的循环方法，这个私有的循环方法调用的是通用的 `loopable.loop()`，而不是更加具体的 `animate()`方法。

9.2.10 HttpMessagePusher

`HttpMessagePusher` 类负责编码和将消息上传到服务器，并解析和存储所有响应。

```

package com.croftsoft.core.net.http.msg;

import java.io.*;
import java.net.*;

import com.croftsoft.core.io.Encoder;

```

```

import com.croftsoft.core.io.Parser;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.net.http.HttpLib;
import com.croftsoft.core.util.loop.FixedDelayLoopGovernor;
import com.croftsoft.core.util.loop.Looper;
import com.croftsoft.core.util.queue.Queue;

public final class HttpMessagePusher
    implements Lifecycle
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

```

类 `HttpMessagePusher` 实现了接口 `Lifecycle`，这样当游戏暂停的时候，网络通信就可以被临时挂起。

```

public static final long      MINIMUM_DELAY      = 100;

public static final String    THREAD_NAME        = "HttpMessagePusher";

public static final int       THREAD_PRIORITY    = Thread.MIN_PRIORITY;

public static final boolean   USE_DAEMON_THREAD = true;

```

100 毫秒的 `MINIMUM_DELAY` 可以防止服务器被客户端请求所压垮。

```

private final Queue    outgoingQueue;

private final Queue    incomingQueue;

private final Encoder  encoder;

private final Looper   looper;

private final URL      url;

private final String   userAgent;

private final String   contentType;

private final Parser   parser;

```

`HttpMessagePusher` 将 `outgoingQueue` 队列中的那些消息上传给服务器，并在 `outgoingQueue` 队列中存储来自于服务器的响应。`Encoder` 负责将发送的消息进行编码。`Looper` 管理网络循环。`url` 是服务器的地址。`userAgent` 是客户机标识。`contentType` 指定发送消息的格式。`parser` 解析服务器的响应。

```

public HttpMessagePusher (
    Queue    outgoingQueue,
    Queue    incomingQueue,
    URL      url,

```



```

String userAgent,
String contentType,
Encoder encoder,
Parser parser )
////////////////////////////////////
{
    NullPointerException.check ( this.outgoingQueue = outgoingQueue );

    NullPointerException.check ( this.incomingQueue = incomingQueue );

    NullPointerException.check ( this.userAgent      = userAgent      );

    NullPointerException.check ( this.contentType   = contentType   );

    NullPointerException.check ( this.encoder       = encoder       );

    NullPointerException.check ( this.parser        = parser        );

    NullPointerException.check ( this.url           = url           );

    loop = new Loop (

        new Loopable ( )
        {
            public boolean loop ( )
            {
                return HttpMessagePusher.this.loop ( );
            }
        },
        new FixedDelayLoopGovernor ( MINIMUM_DELAY, 0 ),
        null,
        THREAD_NAME,
        THREAD_PRIORITY,
        USE_DAEMON_THREAD );
}

```

构造函数创建了 `Loopable` 的一个匿名内部类的实现，该实现委托给 `HttpMessagePusher` 中私有的循环方法。这层间接关系可以防止 `HttpMessagePusher` 提供公有的 `loop()` 方法。

```

public void init ( )
////////////////////////////////////
{
    loop.init ( );
}

[...]

```

4 个生命周期方法都简单地委托给 `loop`。

```

private boolean loop ( )
////////////////////////////////////
{
    try

```

```

{
    Object request = outgoingQueue.pull ( );

    Object response = HttpLib.post (
        url,
        encoder.encode ( request ),
        userAgent,
        contentType,
        parser );

    if ( response != null )
    {
        incomingQueue.append ( response );
    }
}
catch ( InterruptedException ex )
{
}
catch ( Exception ex )
{
    ex.printStackTrace ( );

    return false;
}

return true;
}

```

私有方法 `loop()` 在 `outgoingQueue.pull()` 上被阻塞，直到可以获得一个对象为止。然后再将获得的对象编码，最后再发送给服务器。来自服务器的响应(如果有的话)会被解析并添加到 `incomingQueue` 中。

9.3 游戏特定的客户端代码

到此为止，本章中给出的所有类都是来自于包 `com.croftsoft.core` 的子包中那些具有高通用性和可重用性的类。但是本节要介绍的这个类实现了应用程序特定的客户端游戏代码。

`com.croftsoft.ajgp.http` 包中的类 `ScoreApplet` 是建立服务器连接，获取和更新最高得分的客户端软件。

```

package com.croftsoft.ajgp.http;

[... ]

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.animation.AnimatedApplet;
import com.croftsoft.core.animation.AnimationInit;
import com.croftsoft.core.animation.animator.TextAnimator;
import com.croftsoft.core.animation.painter.ColorPainter;
import com.croftsoft.core.io.StringCoder;

```



```

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.net.http.msg.HttpMessagePusher;
import com.croftsoft.core.util.queue.ListQueue;
import com.croftsoft.core.util.queue.Queue;
public final class ScoreApplet
    extends AnimatedApplet
    //////////////////////////////////////
    //////////////////////////////////////
{
    [...]

```

```
private static final int DELTA_X = 1;
```

```
private static final int DELTA_Y = 1;
```

得分和最高得分将会水平方向上以每帧 DELTA_X 像素,在垂直方向上以每帧 DELTA_Y 像素的速度滑过屏幕。

```
private static final String CHAR_SET_NAME = StringCoder.UTF_8;
```

```
private static final String CONTENT_TYPE = "text/plain";
```

```
private static final String DEFAULT_HOST = "localhost";
```

```
private static final String SERVLET_PATH = "servlet";
```

```
private static final String DEFAULT_PATH = "/score/" + SERVLET_PATH;
```

```
private static final int DEFAULT_PORT = 8080;
```

```
private static final String REQUEST_GET = "get";
```

```
private static final String REQUEST_SET = "set ";
```

```
private static final String USER_AGENT = "Score/1.0";
```

联网常量的值必须与服务器的需求相匹配。

```
private final TextAnimator textAnimator;
```

```
private final Queue outgoingQueue;
```

```
private final Queue incomingQueue;
```

```
//
```

```
private HttpMessagePusher httpMessagePusher;
```

```
private long score;
```

```
private long highScore;
```

```
private boolean gameOver;
```

设置了布尔标志 `gameOver` 以后,就会触发联网代码以更新服务器上的最高得分记录。

```

public static void main ( String [ ] args )
////////////////////////////////////
{
    launch ( new ScoreApplet ( ) );
}

```

除了在浏览器上以 applet 的形式运行以外，类 ScoreApplet 还可以以可执行 JAR 桌面应用程序的形式启动。

[...]

```

public ScoreApplet ( )
////////////////////////////////////
{
    super ( createAnimationInit ( ) );

    textAnimator = new TextAnimator ( );

    textAnimator.setDeltaX ( DELTA_X );

    textAnimator.setDeltaY ( DELTA_Y );

    outgoingQueue = new ListQueue ( );

    incomingQueue = new ListQueue ( );
}

```

不带参数的构造函数初始化 final 类型的实例变量。

```

public void init ( )
////////////////////////////////////
{
    super.init ( );

    addComponentPainter ( new ColorPainter ( ) );

    addComponentAnimator ( textAnimator );
}

```

有些初始化任务必须等到调用超类的 init() 方法以后才可以进行。

```

URL codeBaseURL = null;

try
{
    codeBaseURL = getCodeBase ( );
}
catch ( Exception ex )
{
}

URL servletURL = null;

```



```

try
{
    if ( codeBaseURL != null )
    {
        servletURL = new URL ( codeBaseURL, SERVLET_PATH );
    }
    else
    {
        servletURL = new URL (
            "http", DEFAULT_HOST, DEFAULT_PORT, DEFAULT_PATH );
    }
}
catch ( MalformedURLException ex )
{
    ex.printStackTrace ( );
}

```

我喜欢将游戏客户端 applet 和游戏 servlet 在同一个 WAR 文件中打包。在这种情况下，applet 的 codeBaseURL 将来源于与 servlet 相同的这个服务器，只是路径有些不同。例如，如果 applet 是在 `http://localhost:8080/score/` 位置上，servlet 的路径将是 `http://localhost:8080/score/servlet`。

如果 ScoreApplet 是以桌面应用程序的形式而不是 applet 的形式运行的，调用 `getCodeBase()` 就会抛出异常，codeBaseURL 将保持 null 值。在这种情况下，就会创建一个默认的 servletURL。

```
StringCoder stringCoder = new StringCoder ( CHAR_SET_NAME );
```

```

httpMessagePusher = new HttpMessagePusher (
    outgoingQueue,
    incomingQueue,
    servletURL,
    USER_AGENT,
    CONTENT_TYPE,
    stringCoder,
    stringCoder );

```

```
httpMessagePusher.init ( );
```

```
outgoingQueue.append ( REQUEST_GET );
```

创建并初始化 `httpMessagePusher`。将从服务器上持久存储器中获取最高分的一个请求追加到 `outgoingQueue` 队列中。这个请求直到 `httpMessagePusher` 启动以后才会上传。

```

animatedComponent.addMouseListener (
    new MouseAdapter ( )
    {
        public void mousePressed ( MouseEvent mouseEvent )
        {
            gameOver = true;
        }
    } );
}

```

只要按下鼠标，就会设置 `gameOver` 标志。

```
public void start ( )
///////////////////////////////////////////////////
{
    super.start ( );
    httpMessagePusher.start ( );
}

[...]
```

`start()`方法首先调用超类的方法启动动画。然后再启动 `httpMessagePusher` 激活网络连接。`stop()`方法和 `destroy()`相似，只是它们调用 `httpMessagePusher` 以及其超类相应方法的顺序相反。通常而言，我更加喜欢保持用户界面的活动状态——无论何时进行联网更新，它都是活动的。

```
public void update ( JComponent component )
///////////////////////////////////////////////////
{
    super.update ( component );

    String highScoreString = ( String ) incomingQueue.poll ( );

    if ( highScoreString != null )
    {
        try
        {
            highScore = Long.parseLong ( highScoreString );
        }
        catch ( NumberFormatException ex )
        {
            System.err.println (
                "Unexpected server response: " + highScoreString );
        }
    }
}
```

动画的 `update()`方法会论询 `incomingQueue` 队列中从服务器上下载的所有响应消息。不管队列中是否有消息，`poll()`方法都会立即返回。因为接收到的服务器消息都由 `update()`方法进行排队以等待进行处理，还因为事件分派线程的 `update()`方法是串行执行的，所以不必考虑消息处理代码是否具有线程安全的。

例如，`highScore` 实例变量是以 `long` 类型的形式存储的。与其他基本类型和对象的赋值不同，64 位 `long` 类型的赋值不保证其赋值的原子性。如果在给这个变量赋予一个从服务器下载的新值时，另外一个线程也正试图访问它，就可能产生一些问题。如果在主线程内，接收到的网络消息立即得到处理，而不是排队进行串行处理，那么在进行赋值的时候，在某一单帧上可能会偶然瞬时地显示一个被损坏的值。

```
if ( gameOver )
{
    gameOver = false;

    if ( score > highScore )
```



```

    {
        highScore = score;

        outgoingQueue.append ( REQUEST_SET + highScore );
    }

    score = 0;
}
else
{
    score++;
}

textAnimator.setText (
    "Score: " + score + " High Score: " + highScore );
}

```

游戏结束的时候，新产生的 `score` 就与从服务器上获取的 `highScore` 进行比较。如果新产生的 `score` 比 `highScore` 还大，就会在本地和服务器上同时更新 `highScore` 的值。用 `highScore` 的新值更新服务器的请求也被追加到 `outgoingQueue` 队列，以便于被一个单独的线程上传，这样所有的网络问题或网络延迟都不会阻塞动画线程和锁死用户界面。

9.4 可重用的服务器端代码

很多可重用的客户端代码类也都可以服务器上使用。这些类包括 `Encoder`、`Parser`、`StreamLib` 和 `StringCoder`。在这个示例中，像 `Queue` 和 `Looper` 这样的类就没有在服务器上使用，因为我们期望服务器能够并发地处理多个请求。本节要介绍的新类包括 `Server` 类和 `HttpGatewayServlet` 类。

9.4.1 Server

只要需要一个指向处理请求并返回响应的对象的通用引用，就会使用 `com.croftsoft.core.role` 包中的 `Server` 接口。使用时，我希望将可重用的 HTTP 代码与游戏特定的服务器代码分隔开来。`Server` 接口提供了抽象层。

```

package com.croftsoft.core.role;

public interface Server
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

    public Object serve ( Object request );
}

```

如果使用的是异步消息传递，`serve()` 方法可能会存储请求(随后再进行处理)，并立即返回 `null` 或接收到消息的一个简单确认。在同步消息传递中，请求是在调用线程内立即处理的，并且在可以继续请求以前返回一个响应。

9.4.2 HttpGatewayServlet

`com.croftsoft.core.servlet` 包中的抽象类 `HttpGatewayServlet`，担当从网络传来的 HTTP 请求与在虚拟机器内面向对象的方法调用之间的一个网关的角色。

```
package com.croftsoft.core.servlet;

import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.croftsoft.core.io.Encoder;
import com.croftsoft.core.io.Parser;
import com.croftsoft.core.role.Server;

public abstract class HttpGatewayServlet
    extends HttpServlet
    //////////////////////////////////////
    //////////////////////////////////////
    {
```

抽象类 `HttpGatewayServlet` 扩展了 J2EE 可选包 `javax.servlet.http` 中的类 `HttpServlet`。子类会扩展 `HttpGatewayServlet`，以便提供游戏特定的功能。

```
    private final Server server;

    private final Parser parser;

    private final Encoder encoder;
```

`HttpGatewayServlet` 使用 `Parser` 解析接收到的字节流，并将解析后的请求对象传递给一个 `Server` 实例，然后再使用 `Encoder` 对返回的对象进行编码——如果有返回对象的话。

```
protected HttpGatewayServlet (
    Server server,
    Parser parser,
    Encoder encoder )
    //////////////////////////////////////
    {
        this.server = server;

        this.parser = parser;

        this.encoder = encoder;
    }
```

通过以传递给保护类型的主构造函数的参数的形式提供 `server`、`parser` 和 `encoder` 的不同实现，就可以在这个类中重新使用 HTTP 转发代码的同时，提供特定的功能。例如，对 `parser` 实现使用 `StringCoder`、`SerializableCoder` 或 `XmlCoder`，可以分别接收客户端纯文本命令、压缩的串行化对象或 XML 格式的请求。


```

public HttpGatewayServlet ( )
///////////////////////////////////////////////////
{
    this ( ( Server ) null, ( Parser ) null, ( Encoder ) null );
}

```

公有的无参数构造函数为主构造函数提供了一个 `null` 值。如果不重写下面这几个方法，就可能会导致 `servlet` 抛出异常。

```

protected byte [ ] encode ( Object object )
    throws IOException
///////////////////////////////////////////////////
{
    return encoder.encode ( object );
}

protected Object parse (
    InputStream inputStream,
    int          contentLength )
    throws IOException
///////////////////////////////////////////////////
{
    return parser.parse ( inputStream, contentLength );
}

protected Object serve ( Object request )
///////////////////////////////////////////////////
{
    return server.serve ( request );
}

```

如果 `encoder`、`parser` 或 `server` 仍然为空，使用了该 `final` 实例变量的相应方法就应该被重写。理想情况下，应该能够以构造函数参数的形式提供所有这些变量。但是，有时可能直到调用了 `servlet` 的 `init()` 方法以后，才让这些变量可用。

```

public final void service (
    HttpServletRequest httpRequest,
    HttpServletResponse httpResponse )
    throws IOException, ServletException
///////////////////////////////////////////////////
{

```

通过继承得来的 `HttpServlet` 的方法 `service()` 通常会被委托给 `doGet()` 或 `doPost()`，具体委托给二者中的哪一个，要取决于消息是使用 HTTP GET 命令还是使用 HTTP POST 命令发送的。但是这个方法总会被重写，以便处理这里的消息，而不管这个消息是使用 GET 发送的还是使用 POST 发送的。`HttpGatewayServlet` 中的 `service()` 方法被标记为 `final`，因为它不应该被游戏特定的子类重写。

```

try
{
    Object request = null;

```

```

try
{
    request = parse (
        httpRequest.getInputStream ( ), -1 );
}
catch ( IOException ex )
{
    httpResponse.setStatus (
        HttpServletResponse.SC_BAD_REQUEST );
}

```

servlet 的 `service()` 方法会试图将 HTTP 请求解析成一个消息对象。如果解析不成功, servlet 就会向客户机返回一个错误状态码 `SC_BAD_REQUEST`。

```

if ( request != null )
{
    Object response = serve ( request );

```

如果成功解析了 `request`, 就会再将它传递给 `serve()` 方法进行处理。`serve()` 方法具体会怎样处理这个 `request` 以及要返回什么内容, 都取决于以构造函数参数的形式传递到这类中的 `server` 实现, 或者取决于在这个游戏特定子类中重写的 `serve()` 方法的实现。

```

if ( response != null )
{
    httpResponse.setStatus ( HttpServletResponse.SC_OK );

    byte [ ] bytes = encode ( response );

    httpResponse.getOutputStream ( ).write ( bytes );
}

```

如果这个 `serve()` 方法返回了一个 `response`, 那么状态代码就会被设置为 `OK`, 并且还会以编码的字节流的形式向客户机返回这个 `response` 对象。注意, `server` 还可能会产生一个 `response` 对象, 该对象提示游戏客户端在应用程序层上发生了某种错误, 例如 `BadPasswordError` 错误。但是, 在这个 HTTP 网关传输层, 任何 `response` 都会被认为是 `SC_OK`。

```

else
{
    httpResponse.setStatus (
        HttpServletResponse.SC_ACCEPTED );
}
}

```

如果 `server` 没有返回 `response`, 那么状态代码就会被设为 `SC_ACCEPTED`, 并且不返回任何内容。当 `request` 不需要 `response`, 或 `request` 已经被排队, 以便后面由 `server` 进行处理时, 就会发生这种情况。

```

}
catch ( Exception ex )
{

```



```

        httpServletResponse.setStatus (
            HttpServletResponse.SC_INTERNAL_SERVER_ERROR );

        log ( ex.getMessage ( ), ex );
    }
}

```

如果服务器抛出了一个没有被捕获的异常，那么就会在这里捕获并记录这个异常。客户端会接收到这个响应状态码 SC_INTERNAL_SERVER_ERROR。

9.5 游戏特定的服务器端代码

在这个简单的示例中，只有一个游戏特定的服务器端类 ScoreServlet。如果服务器端代码过于复杂，可能就会希望能将它分为多个类，包括从 Server 中分离出 servlet。

com.croftsoft.ajgp.http 包中的类 ScoreServlet 处理客户端请求以获取或设置最高分记录。这个 ScoreServlet 类重写了其超类的多个方法，包括无参数的构造函数 init()、serve() 和 destroy()。

```

package com.croftsoft.ajgp.http;

import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.io.Encoder;
import com.croftsoft.core.io.Parser;
import com.croftsoft.core.io.SerializableLib;
import com.croftsoft.core.io.StringCoder;
import com.croftsoft.core.role.Server;
import com.croftsoft.core.servlet.HttpGatewayServlet;

public final class ScoreServlet
    extends HttpGatewayServlet
{
    ///////////////////////////////////////////////////////////////////
    ///////////////////////////////////////////////////////////////////

    [...]

    private static final String PRIMARY_FILENAME = "score.dat";

    private static final String BACKUP_FILENAME = "score.bak";

    private static final String CHAR_SET_NAME = StringCoder.UTF_8;

    private static final String GET_REQUEST = "get";

    private static final String SET_REQUEST = "set ";

```

```
private static final StringCoder STRING_CODER
    = new StringCoder ( CHAR_SET_NAME );
```

```
//
```

```
private Long highScoreLong;
```

在初始化过程中，会从以 PRIMARY_FILENAME 形式存储的持久数据文件中，将最高得分读入内存，并以 highScoreLong 的形式进行存储。

```
public ScoreServlet ( )
    //////////////////////////////////////
{
    super ( ( Server ) null, STRING_CODER, STRING_CODER );
}
```

由于无参数构造函数的实现没有向其超类的构造函数提供 Server 参数，所有必须重写超类的 serve() 方法以防止产生 NullPointerException 异常。在这里，就没有提供 Server 的实现，因为 ScoreServlet 将会在它自己的 serve() 方法内直接处理这个请求。

```
[...]
```

```
public void init ( )
    throws ServletException
    //////////////////////////////////////
{
    System.out.println ( SERVLET_INFO );

    try
    {
        highScoreLong = ( Long )
            SerializableLib.load ( PRIMARY_FILENAME, BACKUP_FILENAME );
    }
    catch ( FileNotFoundException ex )
    {
    }
    catch ( Exception ex )
    {
        log ( ex.getMessage ( ), ex );
    }

    if ( highScoreLong == null )
    {
        highScoreLong = new Long ( 0 );
    }
}
```

为了使调试和手动编辑变得更加容易，如果数据量很小并且数据的格式比较简单的话，以纯文本文件的形式保存持久数据将会更加理想。由于持久数据在数据大小和数据复杂性上都会有不断地增加，所以可以使用 SerializableLib 方法，以压缩串行化对象图的形式保存数据，就像在上面这个 init() 方法中演示的一样。


```

public void destroy ( )
////////////////////////////////////
{
    try
    {
        SerializableLib.save (
            highScoreLong, PRIMARY_FILENAME, BACKUP_FILENAME );
    }
    catch ( Exception ex )
    {
        log ( ex.getMessage ( ), ex );
    }
}

```

在这个简单的示例中，直到调用了 servlet 的 `destroy()` 方法以后，才保存最新的最高得分数据。在关闭 servlet 容器的时候，或 servlet 被新版本的代码替换的时候，常常就会发生这种事情。如果在更新客户端以后，调用 `destroy()` 方法以前，服务器崩溃，那么更新就会丢失。为了避免这种事情发生，每次收到更新请求的时候，都会重写数据文件，或者使用一种更加可靠的数据持久性技术，例如 JDBC、实体 EJB 或 JDO。

```

protected Object serve ( Object request )
////////////////////////////////////
{
    String requestString
        = ( ( String ) request ).trim ( ).toLowerCase ( );

    if ( requestString.equals ( GET_REQUEST ) )
    {
        return highScoreLong;
    }

    if ( requestString.startsWith ( SET_REQUEST ) )
    {
        String newHighScoreString
            = requestString.substring ( SET_REQUEST.length ( ) );

        long newHighScore = Long.parseLong ( newHighScoreString );

        synchronized ( this )
        {
            if ( newHighScore > highScoreLong.longValue ( ) )
            {
                highScoreLong = new Long ( newHighScore );
            }
        }

        return null;
    }

    throw new IllegalArgumentException ( );
}

```

由于所有的 HTTP 操作都是在可重用超类 `HttpGatewayServlet` 的 `service()` 方法中处理的, 所以其子类可以使用对象而不使用字节流处理游戏服务器逻辑。在这里, `serve()` 方法就必须实现超出其超类 `parse()` 方法所提供的一些附加解析功能, 以便从一组请求消息中查找最高得分。如果服务器内存中的最高得分超过客户机的请求值, 那么客户机的这一组请求就会被服务器忽略。

最高得分的更新代码被封装在一个同步块中, 以便保证在比较和赋值的操作过程中, 不会有另一个并发的线程对这个值进行更改。如果有两个或更多的客户机同时向 `servlet` 发送请求, 就会发生这种事情。

9.6 打包 WAR

在本章的前面, 提到了我喜欢将 `applet` 客户端 `JRA` 打包在包含了 `servlet` 的同一个 `WAR` 中。由于无符号 `applet` 的安全沙箱限制阻止它与除从其上下载这个 `applet` 的服务器(即代码库服务器)以外的任何服务器进行联系, 所以游戏客户端和游戏服务器放在一起才变得很有意义。

9.6.1 web.xml

`WAR` 的 `/WEB-INF` 目录下的 `web.xml` 文件告诉 `servlet` 容器哪一个类会被作为 `servlet`, 以及应该使用哪一个路径。我通常喜欢将 `servlet` 放在路径 `servlet` 之外, 例如 `http://localhost:8080/score/servlet`。下面就是一个示例 `web.xml` 文件, 可以将这个 `web.xml` 文件作为网络游戏的一个原始模板。

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <servlet>
    <servlet-name>
      score
    </servlet-name>
    <servlet-class>
      com.croftsoft.ajgp.http.ScoreServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>
      score
    </servlet-name>
    <url-pattern>
      /servlet
    </url-pattern>
```



```
</servlet-mapping>
```

```
</web-app>
```

9.6.2 build.xml

Ant 构建文件 build.xml 是用来装配 WAR 文件的。下面的代码演示了这种构建的步骤，我常常使用这种构建步骤构建网络游戏。

```
<target name="score" depends="score_prep,init2">
  <javac srcdir="${src_dir}" destdir="${tmp_dir}">
    <include name="com/croftsoft/ajgp/http/ScoreApplet.java"/>
  </javac>
```

首先编译客户端代码。

```
<jar
  basedir="${tmp_dir}"
  destfile="${arc_dir}/score.jar"
  manifest="${res_dir}/ajgp/score/manifest.txt"
  update="false"/>
```

通常将它与清单中的 Main-Class 入口一起打包在同一个 JAR 文件中，这样它就既可以以桌面应用程序的形式启动，还可以以浏览器 applet 的形式启动。

```
<delete dir="${tmp_dir}"/>
<property name="classes_dir" value="${tmp_dir}/WEB-INF/classes"/>
<mkdir dir="${classes_dir}"/>
<javac
  srcdir="${src_dir}"
  destdir="${classes_dir}"
  classpath="${j2ee_jar}">
  <include name="com/croftsoft/ajgp/http/ScoreServlet.java"/>
</javac>
```

然后再将服务器端的代码编译到特定的 /WEB-INF/classes 目录下。

```
<copy
  overwrite="true"
  file="${arc_dir}/score.jar"
  todir="${tmp_dir}"/>
<copy
  overwrite="true"
  file="${res_dir}/ajgp/score/index.html"
  todir="${tmp_dir}"/>
```

将 applet 客户端 JAR 文件和包含该 applet 的 Web 页面添加到将成为 WAR 根路径的目录中。然后使用一个 URL，例如 <http://localhost:8080/score/>，就可以访问这个 applet 的 Web 页面。

```
<copy
  overwrite="true"
  file="${res_dir}/ajgp/score/WEB-INF/web.xml"
  todir="${tmp_dir}/WEB-INF"/>
```

再将 web.xml 文件添加到特定的/WEB-INF 目录中。

```
<jar
  basedir="${tmp_dir}"
  destfile="${arc_dir}/score.war"
  update="false"/>
<delete dir="${tmp_dir}"/>
</target>
```

最后再将包含 applet 客户端 JAR 文件的 WAR 文件、applet 的 Web 页面、servlet 类和 web.xml 文件装配在一起。

9.7 小结

由于无论是在办公室还是在家中上网，网络防火墙都越来越普遍，所以 HTTP 隧道的使用是现在网络游戏程序所必需的。重用本章给出的这些灵巧的 HTTP 网络库将会确保游戏的联网代码在几乎所有的部署环境中，包括小型的无符号浏览器 applet 中，都可以正常运行。通过使用一个单独的线程，将发送消息在队列中排队上传给服务器，可以防止在网络关闭的时候用户界面死锁。将从服务器上下载接收消息在队列中排队，以便在 update() 方法内进行串行处理，可以防止线程并发所引起的错误。由于无符号 applet 只能与从其中下载该 applet 的服务器建立网络连接，所以将 applet 和 servlet 一起在同一个 Web 应用程序的发布文件中打包就很有意义。本章还给出了一个使用网络库进行持久的服务器端数据存储的无符号 applet 示例。

9.8 参考文献

Coward, Danny. *Java Servlet API Specification, Version 2.3*. Sun Microsystems, 2001.

Hughes, Merlin et al. *Java Network Programming: A Complete Guide to Networking, Streams, and Distributed Computing*, 2nd edition. Greenwich, CT: Manning Publications Co., 1999.

Waldo, Jim et al. *A Note on Distributed Computing*. Sun Microsystems, 1994.



第 10 章 HTTP 轮询机制

一份耕耘，一份收获。
——本杰明·富兰克林

在多玩家联网游戏中，多个玩家在一个共享的虚拟游戏世界里交互。这也意味着服务器上的游戏模型必须镜像到客户端视图中。这样镜像的一个简单方法是客户端周期性地在服务器上轮询游戏状态的最新快照。当使用 HTTP 轮询(polling)技术来完成这种工作的时候，网络将会非常健壮，因为 HTTP 轮询可以穿越防火墙、可以在无符号 applet 的安全限制内运行，并且在网络出现暂时性故障而又恢复通信以后，它还可以立即建立同步。

第 9 章已经介绍了网络开发的配置和 applet-servlet 的通信问题。本章将介绍使用 HTTP 轮询机制的多玩家联网游戏的一个示例程序的源代码。这里介绍的可重用类为第 11 章奠定了基础，第 11 章将介绍网络同步的另一种方法。

10.1 测试示例程序

本章的示例游戏是在第 7 章中介绍的那个单机版游戏 Mars 的多玩家联网版。这个多玩家联网版的游戏，不是在躲避由 AI 控制的敌方坦克的过程中摧毁障碍物，而是让一个玩家与另一玩家战斗。由于在开发这个游戏的单机版的过程中，我非常细心地将模型从视图和控制器中分隔开，所以向多玩家联网版转换的过程就变得相对简单，需要新添加的代码量也大大降低。

```
appletviewer http://localhost:8080/mars/
```

这个游戏所使用的 Ant 的 build.xml 目标是 mars_net_install。上面就是这个 applet 的默认 URL。由于这是一个多玩家联网游戏，需要能够同时启动两个或更多的客户软件，这样才可以观察服务器上的游戏状态是怎样反馈给所有客户机的。每次创建一个新玩家客户端时，就会向虚拟的世界添加一个新坦克，如图 10-1 所示。单击不同的客户端窗口，就可以控制相应玩家的坦克。

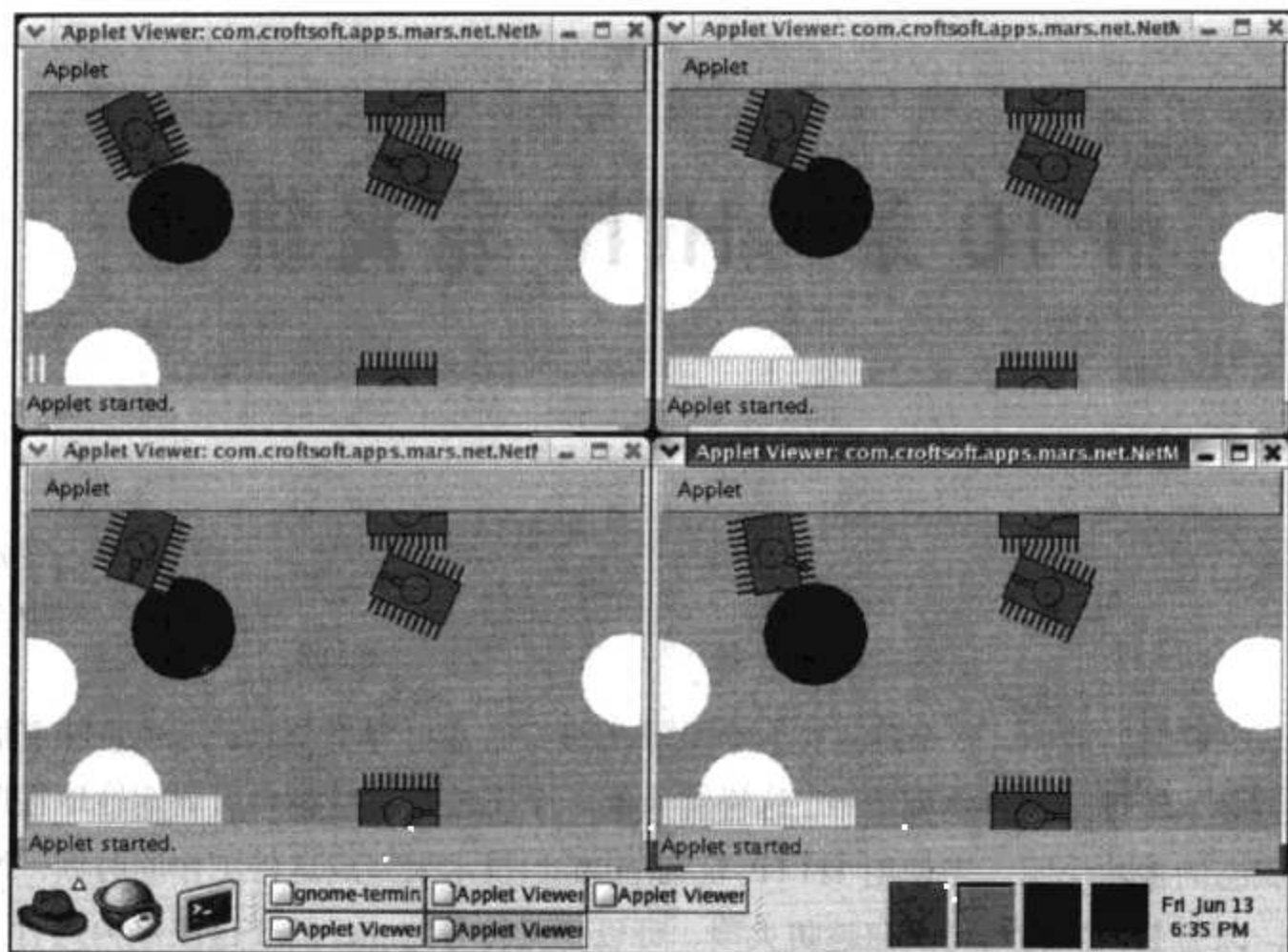


图 10-1 多个玩家客户端和一个共享的虚拟世界

10.2 可重用的客户端代码

在客户端，`SerializableCoder` 类使用对象串行化和压缩，对网络消息进行编码和解析。使用 `HttpMessagePoller` 周期性地从服务器上轮询共享游戏状态的快照。`Consumer` 是为异步使用网络消息的对象提供的一个接口。`QueuePuller` 将从服务器上下载的消息传递给 `Consumer`。`HttpClient` 提供了网络代码的外观(facade)。

10.2.1 SerializableCoder

`SerializableCoder` 类提供了 `Encoder` 和 `Parser` 实现的一个单态实例。它使用 `SerializableLib` 的对象串行化和压缩方法。使用它编码和解析在 Mars 游戏客户端和服务端之间传递的消息。

```
package com.croftsoft.core.io;
```

```
import java.io.*;
```

```
public final class SerializableCoder
```

```
    implements Encoder, Parser
```

```
    {
```

```
    public static final SerializableCoder INSTANCE
```

```
        = new SerializableCoder ( );
```

```
    }
```



```

////////////////////////////////////
public byte [ ] encode ( Object object )
    throws IOException
////////////////////////////////////
{
    return SerializableLib.compress ( ( Serializable ) object );
}

public Object parse (
    InputStream inputStream,
    int          contentLength )
    throws IOException
////////////////////////////////////
{
    try
    {
        return SerializableLib.load ( inputStream );
    }
    catch ( ClassNotFoundException ex )
    {
        throw ( IOException ) new IOException ( ).initCause ( ex );
    }
}

////////////////////////////////////
////////////////////////////////////

private SerializableCoder ( ) { }

```

10.2.2 HttpMessagePoller

com.croftsoft.core.net.http.msg 包中的 HttpMessagePoller 类用来轮询服务器的消息。客户端可以使用在请求之间有着合理延迟的轮询，从服务器上下载周期性的状态快照。

虽然可以使用 HttpMessagePoller 或 HttpMessagePusher 进行轮询，但是 HttpMessagePoller 仍然是完成该任务的首选。在 HttpMessagePoller 中，只会将请求消息编码为字节数组一次，然后再在一个循环中，重复地将请求消息传输给服务器。与 HttpMessagePusher 相比较而言，后者是从发送队列中拖出消息，并在每次上传消息之前都对它们重新进行编码。使用 HttpMessagePusher 进行轮询需要一个单独的循环，这个单独的循环周期性地将请求消息追加到队列。

```

package com.croftsoft.core.net.http.msg;

import java.io.*;
import java.net.*;

import com.croftsoft.core.io.Parser;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.math.MathConstants;
import com.croftsoft.core.net.http.HttpLib;
import com.croftsoft.core.util.loop.FixedDelayLoopGovernor;

```

```
import com.croftsoft.core.util.loop.Looper;
import com.croftsoft.core.util.queue.Queue;

public final class HttpMessagePoller
    implements Lifecycle
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////
}
```

HttpMessagePoller 实现了 Lifecycle, 这样网络通信就可以进行暂停和恢复。它还导入了前面介绍过的多个 CroftSoft 内核类。

```
public static final long    DEFAULT_POLLING_PERIOD_MIN
    = MathConstants.MILLISECONDS_PER_SECOND;

public static final long    DEFAULT_POLLING_PERIOD_MAX
    = MathConstants.MILLISECONDS_PER_DAY;

public static final long    DEFAULT_POLLING_PERIOD_INIT
    = DEFAULT_POLLING_PERIOD_MIN;

public static final double  DEFAULT_POLLING_PERIOD_MULT
    = 1.1;

public static final double  DEFAULT_POLLING_PERIOD_DIVI
    = 1.1;

public static final long    DEFAULT_POLLING_PERIOD_INCR
    = MathConstants.MILLISECONDS_PER_SECOND;
```

HttpMessagePoller 提供自适应轮询机制。在自适应轮询中, 客户机试图优化请求之间的延迟。当来自于服务器的响应不正确或响应为空时, 自适应的轮询算法就会增加轮询的延迟。当响应是正确的, 并包含一个有效消息时, 它就会减少轮询的延迟。当客户机试图和一个临时不能提供服务的服务器进行连接的时候, 这对降低网络流量非常有用。客户机起先会频繁地轮询服务器, 然后会逐渐减少它尝试的次数, 因为它意识到自己并没有得到响应。

默认的最小值和最初的轮询周期是 1 秒。默认的最大轮询周期是一天。如果服务器关闭了, 或者是服务器没有什么信息要提供给客户机, 客户机就会开始每秒轮询服务器一次, 但是会慢慢降低它的轮询频率, 直到降低到每天只轮询一次以后就不再降低其轮询频率了。如果服务器恢复了, 或它又开始给客户机提供有效的消息, 轮询的频率就会又慢慢地增加。默认情况下, 轮询周期增加延迟的因数是 1.1, 这就意味着轮询周期的增加幅度是 10%。轮询周期降低延迟的因数也是 1.1。改变轮询周期最小的默认增量是 1 秒。

```
public static final String  THREAD_NAME = "HttpMessagePoller";

public static final int    THREAD_PRIORITY = Thread.MIN_PRIORITY;

public static final boolean USE_DAEMON_THREAD = true;
```

如果要将 HttpMessagePoller 用在一个没有运行其他线程的环境中, 最好将 USE_DAEMON_THREAD 修改为 false, 这样可以防止程序在启动以后立即终止。


```

[...]
```

```

private final long    pollingPeriodMin;

private final long    pollingPeriodMax;

private final double  pollingPeriodMult;

private final double  pollingPeriodDivi;

private final long    pollingPeriodIncr;

//

private byte [ ] requestBytes;

private long    pollingPeriod;

```

自适应的轮询周期参数是在主构造函数内以 `final` 变量的形式设置的。如果想使用一个不用调整的恒定轮询周期，将 `pollingPeriodMin` 和 `pollingPeriodMax` 设置为同一个值就可以了。

```

public HttpMessagePoller (
    URL        url,
    String      userAgent,
    String      contentType,
    byte [ ] requestBytes,
    Parser      parser,
    Queue       incomingQueue,
    long        pollingPeriodMin,
    long        pollingPeriodMax,
    long        pollingPeriodInit,
    double      pollingPeriodMult,
    double      pollingPeriodDivi,
    long        pollingPeriodIncr )
////////////////////////////////////
{
    [...]
}

[...]
```

除了增加了自适应的轮询周期参数以外，`HttpMessagePoller` 的构造函数与第 9 章介绍的 `HttpMessagePusher` 的构造函数非常相似。

```

public void setRequestBytes ( byte [ ] requestBytes )
////////////////////////////////////
{
    NullPointerException.check ( this.requestBytes = requestBytes );
}

```

为了能够不断地发送同一个请求对 `HttpMessagePoller` 进行了一定的优化处理。为此，通常是一次性地将请求转换为字节数组，然后就不再进行更改了。而 `HttpMessagePusher` 在每次有

请求从发送队列中拖出以后，都要经历一个编码过程。但是如果这样做的话，mutator 方法 setRequestBytes()就允许在调用构造函数以后更改请求。

```
public synchronized long setPollingPeriod ( long pollingPeriod )
///////////////////////////////////////////////////
{
    if ( pollingPeriod < pollingPeriodMin )
    {
        pollingPeriod = pollingPeriodMin;
    }

    if ( pollingPeriod > pollingPeriodMax )
    {
        pollingPeriod = pollingPeriodMax;
    }

    this.pollingPeriod = pollingPeriod;

    looper.setLoopGovernor (
        new FixedDelayLoopGovernor ( pollingPeriod, 0 ) );

    return pollingPeriod;
}
```

虽然通常会将这个工作留给自适应轮询算法完成，但是我们还是可以使用公有的 mutator 方法 setPollingPeriod()，显式地改变轮询周期。

```
setPollingPeriod().
public void init ( )
///////////////////////////////////////////////////
{
    looper.init ( );
}

[...]
```

这 4 个生命周期方法简单地委托给 looper。

```
private boolean loop ( )
///////////////////////////////////////////////////
{
    try
    {
        Object response = HttpLib.post (
            url,
            requestBytes,
            userAgent,
            contentType,
            parser );

        if ( response != null )
        {
```



```

        incomingQueue.append ( response );

        decreasePollingPeriod ( );
    }
    else
    {
        increasePollingPeriod ( );
    }

    return true;
}
catch ( Exception ex )
{
    ex.printStackTrace ( );

    increasePollingPeriod ( );

    return true;
}
}

```

私有方法 `loop()` 将请求传递给服务器并存储来自于服务器的响应。如果有一个错误或者没有有效消息可供下载，轮询周期就会增加；否则，就缩短轮询周期。经验告诉我，这个算法会调整轮询延迟，所以它以下面这样的频率进行轮询：大约两倍于消息在服务器上变为可用的速度。例如，如果服务器向客户端提供消息的频率大约是平均每天一次，那么轮询的频率就会大约是每天两次。

```

private void increasePollingPeriod ( )
///////////////////////////////////////////////////////////////////
{
    long newPollingPeriod
        = ( long ) ( pollingPeriod * pollingPeriodMult );

    if ( newPollingPeriod < pollingPeriod + pollingPeriodIncr )
    {
        newPollingPeriod = pollingPeriod + pollingPeriodIncr;
    }

    if ( newPollingPeriod > pollingPeriodMax )
    {
        newPollingPeriod = pollingPeriodMax;
    }

    if ( pollingPeriod != newPollingPeriod )
    {
        if ( DEBUG )
        {
            System.out.println (
                "Increasing polling period from " + pollingPeriod
                + " to " + newPollingPeriod + " milliseconds." );
        }
    }
}

```

```

        loopper.setLoopGovernor (
            new FixedDelayLoopGovernor ( newPollingPeriod, 0 ) );

        pollingPeriod = newPollingPeriod;
    }

    private void decreasePollingPeriod ( )
    //////////////////////////////////////
    {
        [...]
    }

```

私有方法 `increasePollingPeriod()` 和 `decreasePollingPeriod()` 用来调整轮询周期，同时还确保轮询周期在最大值和最小值的范围以内。注意，如果轮询周期为 0，那么将其乘以 `pollingPeriodMult` 因数 1.1 的结果就仍然为 0。增量 `pollingPeriodIncr` 用来确保在这样的类中使用一个递增量或递减量(例如 1 秒)。方法 `decreasePollingPeriod()` 与 `increasePollingPeriod()` 非常相似，只是前者是将当前的 `pollingPeriod` 除以 `pollingPeriodDivi` 得到的。

10.2.3 Consumer

`HttpMessagePoller` 和 `HttpMessagePusher` 从服务器上下载响应和消息，并将它们追加到接收队列中以便客户机稍后进行处理。使用一个单独的线程将这些接收消息从队列中取出并处理，这样与 `HttpMessagePoller` 和 `HttpMessagePusher` 一起使用的两个联网线程就可以单独运行。客户端游戏代码可以为这些新的消息轮询接收队列，或者为异步处理它们提供一个回调接口。

```

package com.croftsoft.core.role;

public interface Consumer
    //////////////////////////////////////
    //////////////////////////////////////
    {

    public void consume ( Object o );

```

对必须接收对象(例如要被处理的一个消息)的所有事物都使用 `Consumer` 作为其回调接口。在包 `com.croftsoft.core.role` 中保存 `Consumer`，因为我相信这个接口为对象定义了一个抽象角色(role)。同一个包和它的子包中的其他接口还包括 `Actor`、`Filter` 和 `Server`。

10.2.4 QueuePuller

使用类 `QueuePuller` 将对象从队列中取出并将它传递给 `Consumer` 进行处理。在这个示例中，`QueuePuller` 将从服务器上下载的消息拖出客户端的接收队列并将这些消息传递给游戏代码，这样就可以更新客户机上的视图了。

```

package com.croftsoft.core.util.queue;

import com.croftsoft.core.lang.NullArgumentException;

```



```

import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.role.Consumer;
import com.croftsoft.core.util.loop.Loopable;
import com.croftsoft.core.util.loop.Looper;

public final class QueuePuller
    implements Lifecycle
    //////////////////////////////////////
    //////////////////////////////////////
    {

```

类 QueuePuller 实现了 Lifecycle，因此它可以被挂起和恢复。

```

private final Queue    queue;

private final Consumer consumer;

private final Looper   loop;

////////////////////////////////////
////////////////////////////////////

public QueuePuller (
    Queue queue,
    Consumer consumer )
    //////////////////////////////////////
    {
        NullPointerException.check ( this.queue = queue );

        NullPointerException.check ( this.consumer = consumer );

        loop = new Looper (
            new Loopable ( )
            {
                public boolean loop ( )
                {
                    return QueuePuller.this.loop ( );
                }
            } );
    }

```

这里使用了 Loopable 的一个匿名内部类实例，这样 QueuePuller 类的 loop() 方法就不再是公有方法。

```

public void init ( )
    //////////////////////////////////////
    {
        loop.init ( );
    }

    [...]

```

4 个生命周期方法只是简单地委托给 loop。

```

public boolean append ( Object o )
///////////////////////////////////////////////////
{
    return queue.append ( o );
}

public Object replace ( Object o )
///////////////////////////////////////////////////
{
    return queue.replace ( o );
}

```

提供了 `append()` 和 `replace()` 这两个便利方法以后，调用代码就不再需要存储对 `queue` 的单独引用。

```

private boolean loop ( )
///////////////////////////////////////////////////
{
    try
    {
        consumer.consume ( queue.pull ( ) );
    }
    catch ( InterruptedException ex )
    {
    }

    return true;
}

```

私有方法 `loop()` 将对象从 `queue` 中拖出并将它们传递给 `consumer`。即使在 `queue.pull()` 上进行等待的同时，可能会阻塞 `looper` 线程，也不会影响其他游戏代码的运行。从 `consumer` 的角度看来，消息的到达都将是异步的。

10.2.5 HttpClient

我习惯将 `HttpMessagePoller`、`HttpMessagePusher` 和 `QueuePuller` 结合在一起使用，因此创建了一个外观类 `HttpClient`，这个类允许将这些类作为一个复合对象进行操作。

```

package com.croftsoft.core.net.http.msg;

import java.net.*;
import java.util.*;

import com.croftsoft.core.io.Encoder;
import com.croftsoft.core.io.Parser;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.role.Consumer;
import com.croftsoft.core.util.queue.ListQueue;
import com.croftsoft.core.util.queue.Queue;

```



```
import com.croftsoft.core.util.queue.QueuePuller;

public final class HttpMessageClient
    implements Lifecycle
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

```

像组件类一样，HttpMessageClient 也实现了 Lifecycle。在 HttpMessageClient 上调用 init()、start()、stop() 或 destroy() 方法将会通过委托给 HttpMessagePoller、HttpMessagePusher 和 QueuePuller 上相应的方法，起到一石三鸟的效果。

```
private final Queue          incomingQueue;

private final Queue          outgoingQueue;

private final HttpMessagePoller httpMessagePoller;

private final HttpMessagePusher httpMessagePusher;

private final QueuePuller     incomingQueuePuller;
```

HttpMessageClient 隐藏了这些实例变量所使用的这 5 个类的基本用法。这简化了游戏客户端软件的设计，因为它只需要与 HttpMessageClient 类进行交互。

```
public HttpMessageClient (
    URL      url,
    [...]
    long     pollingPeriodIncr )
{
    ///////////////////////////////////////////////////

    incomingQueue = new ListQueue ( );

    outgoingQueue = new ListQueue ( );

    httpMessagePoller = new HttpMessagePoller (
        url,
        [...]
        pollingPeriodIncr );

    httpMessagePusher = new HttpMessagePusher (
        outgoingQueue,
        incomingQueue,
        url,
        userAgent,
        contentType,
        encoder,
        parser );

    if ( consumer != null )
    {
        incomingQueuePuller = new QueuePuller ( incomingQueue, consumer );
    }
}
```

```

    }
    else
    {
        incomingQueuePuller = null;
    }
}

```

[...]

主构造函数使用作为构造函数参数传递的初始化值，创建 `incomingQueue`、`outgoingQueue`、`httpMessagePoller`、`httpMessagePusher` 和 `queuePuller` 的实例。如果没有提供 `consumer`，就不会创建 `QueuePuller`。

```
public Queue getIncomingQueue ( ) { return incomingQueue; }
```

```
public Queue getOutgoingQueue ( ) { return outgoingQueue; }
```

对私有队列的直接访问是通过 `getIncomingQueue()` 和 `getOutgoingQueue()` 方法提供的。一般都不需要直接访问，因为通过便利方法进行间接访问也是可以的。

```

public void setRequestBytes ( byte [ ] requestBytes )
////////////////////////////////////
{
    httpMessagePoller.setRequestBytes ( requestBytes );
}

```

由 `HttpMessagePoller` 使用的经过编码的请求可以使用 `mutator` 方法 `setRequestBytes()` 进行更改。通常在构造之后，`requestBytes` 就不会发生改变了。

```

public synchronized void init ( )
////////////////////////////////////
{
    if ( incomingQueuePuller != null )
    {
        incomingQueuePuller.init ( );
    }

    httpMessagePoller.init ( );

    httpMessagePusher.init ( );
}

```

[...]

4 个生命周期方法简单地委托给 `HttpMessagePoller`、`HttpMessagePusher` 以及 `QueuePuller`——如果 `QueuePuller` 可用的话。这样就可以确保这里全部的 3 个联网线程都会一起挂起或重启，这通常也是我们所期望的。

```

public boolean append ( Object o )
////////////////////////////////////
{
    return outgoingQueue.append ( o );
}

```



```

}

public void replace ( Object o )
////////////////////////////////////
{
    outgoingQueue.replace ( o );
}

```

便利方法 `append()` 和 `replace()` 将消息插入 `outgoingQueue`。由于这些方法都可用，所以没有必要使用 `accessor` 方法 `getOutgoingQueue()` 直接访问私有的 `outgoingQueue` 实例变量。

```

public Object poll ( )
////////////////////////////////////
{
    return incomingQueue.poll ( );
}

public Object pull ( )
    throws InterruptedException
////////////////////////////////////
{
    return incomingQueue.pull ( );
}

```

便利方法 `poll()` 和 `pull()` 从 `incomingQueue` 中获取下载的消息。如果 `consumer` 是以构造函数的参数形式给出的，就不再需要这些方法，因为这样 `QueuePuller` 就会是活动的。

10.3 游戏特定的客户端代码

客户端游戏代码中有限的几个可重用的应用程序特定类包括 `Request`、`AbstractRequest`、`FireRequest`、`MoveRequest`、`ViewRequest`、`GameData`、`Synchronizer`、`NetController` 和 `NetMain`。

10.3.1 Request

从客户端发送到服务器的所有请求对象的基本接口就是 `com.croftsoft.apps.mars.net.request` 包中的 `Request`。在这个游戏示例中，它简单地提供发出请求的玩家名称。其更健壮版还将包括用于验证的密码或数字签名。将来甚至可以通过使用 `HTTPS(HTTP over Secure Sockets Layer)` 加密网络通信来提高安全性。更多的信息，请参考 Java 内核包 `javax.net.ssl` 中的类 `HttpsURLConnection`。

```

package com.croftsoft.apps.mars.net.request;

import java.io.Serializable;

public interface Request
    extends Serializable
    //////////////////////////////////////
    //////////////////////////////////////

```

```
{
```

```
public String getPlayerName ( );
```

所有 Request 的实现都是 Serializable 的。

10.3.2 AbstractRequest

抽象类 AbstractRequest 提供了接口 Request 的一个基本实现。

```
package com.croftsoft.apps.mars.net.request;
```

```
public abstract class AbstractRequest
```

```
implements Request
```

```
////////////////////////////////////
////////////////////////////////////
{
```

```
private final String playerName;
```

```
////////////////////////////////////
////////////////////////////////////
```

```
public AbstractRequest ( String playerName )
```

```
////////////////////////////////////
{
this.playerName = playerName;
}
```

```
////////////////////////////////////
////////////////////////////////////
```

```
public String getPlayerName ( ) { return playerName; }
```

虽然很多请求都需要玩家的标识，但是还是有一些请求不需要玩家的标识。在这些不需要玩家标识的请求中，playerName 可能会为 null。

10.3.3 FireRequest

FireRequest 是让玩家坦克炮开火的一个请求。

```
package com.croftsoft.apps.mars.net.request;
```

```
public final class FireRequest
```

```
extends AbstractRequest
```

```
////////////////////////////////////
////////////////////////////////////
{
```

```
private static final long serialVersionUID = 0L;
```



```

////////////////////////////////////
////////////////////////////////////

public FireRequest ( String playerName )
////////////////////////////////////
{
    super ( playerName );
}

```

FireRequest 中没有太多内容，因为需要传送的信息只是要开火的那个坦克对应玩家的名称。

10.3.4 MoveRequest

MoveRequest 是将玩家的坦克移动到给定目标的一个请求。

```

package com.croftsoft.apps.mars.net.request;

import com.croftsoft.core.math.geom.Point2DD;
import com.croftsoft.core.math.geom.PointXY;

public final class MoveRequest
    extends AbstractRequest
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

    //

    private final Point2DD destination;

    //////////////////////////////////////
    //////////////////////////////////////

    public MoveRequest ( String playerName )
    //////////////////////////////////////
    {
        super ( playerName );

        destination = new Point2DD ( );
    }

    //////////////////////////////////////
    //////////////////////////////////////

    public PointXY getDestination ( ) { return destination; }

    public void setDestination (
        double x,
        double y )
    //////////////////////////////////////

```

```

{
    destination.x = x;

    destination.y = y;
}

```

MoveRequest 是可变的，因此通过简单地更新目标就可以重新使用一个实例。

10.3.5 ViewRequest

在将一个已编码的 **ViewRequest** 发送到服务器以后，服务器就会在玩家位置的视图内部，用当前游戏状态的一个快照进行响应。为了简单起见，这个游戏示例中的服务器实现会返回整个游戏世界的游戏状态。

```

package com.croftsoft.apps.mars.net.request;

public final class ViewRequest
    extends AbstractRequest

    //////////////////////////////////////
    //////////////////////////////////////
    {

        private static final long serialVersionUID = 0L;

        //////////////////////////////////////
        //////////////////////////////////////

        public ViewRequest ( String playerName )
        //////////////////////////////////////
        {
            super ( playerName );
        }
    }

```

对于游戏中没有参与游戏的客户机，就应该将 **playerName** 设置为 **null**。在这种情况下，服务器将会用游戏状态的一个快照进行响应，而不为客户机创建一个要进行控制的玩家坦克。

10.3.6 GameData

就像是为客户机上产生的请求专门提供 **Request** 一样，您可能希望为在服务器上产生但又发送给客户机的请求(例如 **Response**、**Message** 或 **Event**)创建一个基本接口。对这个游戏来说，服务器发送的惟一消息就是 **com.croftsoft.apps.mars.net** 包中串行化的游戏状态数据类 **GameData** 的一个实例。

```

package com.croftsoft.apps.mars.net;

import java.io.*;
import java.util.*;

import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.util.NullIterator;

```



```

import com.croftsoft.apps.mars.model.GameAccessor;
import com.croftsoft.apps.mars.model.TankAccessor;
import com.croftsoft.apps.mars.model.WorldAccessor;
import com.croftsoft.apps.mars.model.seri.SeriTank;
import com.croftsoft.apps.mars.model.seri.SeriWorld;

public final class GameData
    implements GameAccessor, Serializable
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

    //

    private final SeriWorld seriWorld;

    private final SeriTank playerSeriTank;

    //////////////////////////////////////
    //////////////////////////////////////

    public GameData (
        SeriWorld  seriWorld,
        SeriTank   playerSeriTank )
    //////////////////////////////////////
    {
        NullPointerException.check ( this.seriWorld = seriWorld );

        this.playerSeriTank = playerSeriTank;
    }

    //////////////////////////////////////
    // interface GameAccessor methods
    //////////////////////////////////////

    public int      getLevel ( ) { return 0; }

    public Iterator getPath ( ) { return NullIterator.INSTANCE; }

    public TankAccessor getPlayerTankAccessor ( )
    //////////////////////////////////////
    {
        return playerSeriTank;
    }

    public WorldAccessor getWorldAccessor ( )
    //////////////////////////////////////
    {
        return seriWorld;
    }
}

```

类 `GameData` 实现了接口 `GameAccessor`，这就是接收终端上的客户端软件所需要的接口。记住，这是服务器上游戏模型的一个快照备份而不是模型本身。任何 `mutator` 方法都只能起到让人混淆的作用。

在 `Mars` 游戏的网络版中，玩家在清除了所有的障碍物以后，并不前进到下一关，而是四处游动并互相射击。为此，`getLevel()` 总是返回 0。如果 `ViewRequest` 中的 `playerName` 为 `null`，相应的 `playerTankAccessor` 就为空，因为客户端只是简单地旁观游戏中的打斗而不参与游戏的打斗。

10.3.7 Synchronizer

以镜像到客户机上的形式，使用 `com.croftsoft.apps.mars.net` 包中的类 `Synchronizer`，将游戏模型的状态与服务器上游戏模型的状态进行同步。还可以使用它封装某个现场上的所有联网代码。

```
package com.croftsoft.apps.mars.net;

[...]

import com.croftsoft.core.io.SerializableCoder;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.lang.lifecycle.Lifecycle;
import com.croftsoft.core.math.MathConstants;
import com.croftsoft.core.net.http.msg.HttpMessageClient;
import com.croftsoft.core.role.Consumer;
import com.croftsoft.core.util.NullIterator;

import com.croftsoft.apps.mars.model.GameAccessor;
import com.croftsoft.apps.mars.model.NullGameAccessor;
import com.croftsoft.apps.mars.model.TankAccessor;
import com.croftsoft.apps.mars.model.WorldAccessor;
import com.croftsoft.apps.mars.net.request.ViewRequest;

public final class Synchronizer
    implements GameAccessor, Lifecycle
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////
}
```

由于客户端视图会使用 `Synchronizer` 作为远程服务器上游戏模型的本地表示，所以类 `Synchronizer` 实现了接口 `GameAccessor`。这之所以成为可能，是因为在这个游戏最原始的单机版中，已经使用接口 `GameAccessor` 将模型与视图类 `GameAnimator` 分隔开了。

`Synchronizer` 也实现了接口 `Lifecycle`，因此联网线程可以被挂起或恢复。当线程被挂起的时候，服务器游戏模型的客户端备份就会变为过期的备份。

```
private static final String DEFAULT_HOST = "localhost";

private static final int    DEFAULT_PORT = 8080;
```



```
private static final String SERVLET_PATH = "servlet";

private static final String DEFAULT_PATH = "/mars/" + SERVLET_PATH;

servlet 的默认 URL 是 http://localhost:8080/mars/servlet。

private static final String USER_AGENT = "CroftSoft Mars";

private static final String CONTENT_TYPE
    = "application/octet-stream";
```

游戏客户端会向服务器将它自己标志为 USER_AGENT。由于消息的格式是应用程序特定的二进制文件格式，所以 CONTENT_TYPE 是以 application/octet-stream 形式给出的。如果使用 XML 进行编码而不是使用对象串行化，那么 CONTENT_TYPE 就将会是 text/xml。

```
private static final long    POLLING_PERIOD_MIN
    = 100;

private static final long    POLLING_PERIOD_MAX
    = MathConstants.MILLISECONDS_PER_DAY;

private static final long    POLLING_PERIOD_INIT
    = POLLING_PERIOD_MIN;

private static final double POLLING_PERIOD_MULT
    = 2.0;

private static final double POLLING_PERIOD_DIVI
    = Double.POSITIVE_INFINITY;

private static final long    POLLING_PERIOD_INCR
    = MathConstants.MILLISECONDS_PER_SECOND;
```

在这个示例中，客户机每隔 100 毫秒就会轮询服务器以下载游戏状态的快照。如果有服务器或网络错误，轮询周期就会立即增加 1 秒。以后，轮询周期在每次尝试失败以后都会翻一倍，直到恢复通信为止。由于 POLLING_PERIOD_DIVI 设为无穷大，所以轮询周期就会立即恢复到最初的 100 毫秒。

```
private final HttpClient httpMessageClient;

//

private GameAccessor gameAccessor;

private int            level;
```

从服务器上下载的游戏状态以 gameAccessor 的形式存储，并通过第 9 章介绍的 GameAnimator 进行访问。记住，当 level 发生改变的时候，GameAnimator 就会在假定不再使用所有原来的模型的前提下，完全刷新它自己。

```
public Synchronizer (
    String playerName,
    URL    codeBaseURL )
```

```

    throws IOException, MalformedURLException
    //////////////////////////////////////
    {
        NullPointerException.check ( playerName );

        URL servletURL = null;

        if ( codeBaseURL != null )
        {
            servletURL = new URL ( codeBaseURL, SERVLET_PATH );
        }
        else
        {
            servletURL
                = new URL ( "http", DEFAULT_HOST, DEFAULT_PORT, DEFAULT_PATH );
        }
    }

```

我喜欢将游戏客户端 applet 和游戏 sevlet 打包在同一个 Web 应用程序归档(WAR)文件中。在这种情况下, applet codeBaseURL 就会与 servlet 来自于同一个服务器,只是路径有一点不同。例如,如果 applet 是在 http://localhost:8080/mars/上,那么 servlet 就会在 http://localhost:8080/mars/servlet 上。

```

        httpMessageClient = new HttpMessageClient (
            servletURL,
            USER_AGENT,
            SerializableCoder.INSTANCE,
            SerializableCoder.INSTANCE,
            CONTENT_TYPE,
            SerializableCoder.INSTANCE.encode (
                new ViewRequest ( playerName ) ),
            new Consumer ( )
            {
                public void consume ( Object o )
                {
                    Synchronizer.this.consume ( o );
                }
            },
            POLLING_PERIOD_MIN,
            POLLING_PERIOD_MAX,
            POLLING_PERIOD_INIT,
            POLLING_PERIOD_MULT,
            POLLING_PERIOD_DIVI,
            POLLING_PERIOD_INCR );

        gameAccessor = NullGameAccessor.INSTANCE;
    }

```

初始的 gameAccessor 是 NullGameAccessor 的一个空对象单态实例。在第一次成功地将游戏状态从服务器上下载下来的时候,就会替换这个实例。

```

    public void replace ( Object o )
    //////////////////////////////////////
    {

```



```

    httpMessageClient.replace ( o );
}

```

除了处理周期性的视图请求以外，还使用 Synchronizer 发送移动玩家坦克和坦克炮开火的请求。方法 `replace()` 被委托给 `httpMessageClient` 中相应的方法。这里使用的是 `replace()` 而不是 `append()`，这一点非常重要，因为与通过网络将它们上传相比，这可以更快地将这些请求添加到队列中。例如，当玩家将鼠标指针滑过屏幕的时候，产生的鼠标事件就会非常多。每一个鼠标事件都会产生一个相应的 `MoveRequest` 请求。但是，只有最近产生的鼠标事件才需要上传到服务器。如果 `FireRequest` 或 `MoveRequest` 已经在发送队列中，就会用新的请求将它替换。

```

public int getLevel ( )
///////////////////////////////////////////////////
{
    return level;
}

public Iterator getPath ( )
///////////////////////////////////////////////////
{
    return NullIterator.INSTANCE;
}

public TankAccessor getPlayerTankAccessor ( )
///////////////////////////////////////////////////
{
    return gameAccessor.getPlayerTankAccessor ( );
}

public WorldAccessor getWorldAccessor ( )
///////////////////////////////////////////////////
{
    return gameAccessor.getWorldAccessor ( );
}

```

这是一些使 Synchronizer 看起来很像 `GameAccessor` 的方法。只要是通过网络周期性地刷新委托的 `gameAccessor` 实例，本地游戏状态就会继续保持，即使游戏实际上是在服务器上运行的。

```

public void init ( )
///////////////////////////////////////////////////
{
    httpMessageClient.init ( );
}

[...]

```

简单地将生命周期方法委托给 `httpMessageClient`。

```

private void consume ( Object o )
///////////////////////////////////////////////////
{

```

```

    gameAccessor = ( GameAccessor ) o;

    level++;
}

```

私有的 `consume()` 方法处理来自于服务器的接收消息。在这个示例中，所有服务器的消息都会实现 `GameAccessor` 接口，这是我们都知的事情。实例变量 `gameAccessor` 负责保存来自于服务器的游戏状态的最新快照。递增游戏关，这样 `GameAnimator` 就可以完全刷新它自己。这种效果不是很明显，但是演示起来比较简单。第 11 章还将介绍一种递增更新机制，这种新的更新机制可以代替这里的这种新机制。

10.3.8 NetController

`NetController` 将玩家的键盘和鼠标输入转换为要上传到游戏服务器以控制玩家坦克的消息。

```

package com.croftsoft.apps.mars.net;

import java.awt.*;
import java.awt.event.*;

import com.croftsoft.core.gui.event.UserInputAdapter;
import com.croftsoft.core.lang.NullArgumentException;
import com.croftsoft.core.math.geom.Point2DD;

import com.croftsoft.apps.mars.net.request.FireRequest;
import com.croftsoft.apps.mars.net.request.MoveRequest;

public final class NetController
    extends UserInputAdapter
    //////////////////////////////////////
    //////////////////////////////////////
    {

    private final String      playerName;

    private final Synchronizer synchronizer;

    private final FireRequest  fireRequest;

    private final MoveRequest  moveRequest;

    //////////////////////////////////////
    //////////////////////////////////////

    public NetController (
        String      playerName,
        Synchronizer synchronizer,
        Component    component )
    //////////////////////////////////////
    {
        NullArgumentException.check ( this.playerName = playerName );
    }
}

```



```

    NullPointerException.check ( this.synchronizer = synchronizer );

    NullPointerException.check ( component );

    component.addMouseListener ( this );

    component.addMouseMotionListener ( this );

    component.addKeyListener ( this );

    component.requestFocus ( );

    fireRequest = new FireRequest ( playerName );

    moveRequest = new MoveRequest ( playerName );
}

```

`fireRequest` 和 `moveRequest` 实例变量被声明为 `final`, 因为它们在构造过程中只被创建一次, 此后就会不断地重复使用。

```

public void keyPressed ( KeyEvent keyEvent )
///////////////////////////////////////////////////////////////////
{
    if ( keyEvent.getKeyChar ( ) == ' ' )
    {
        fire ( );
    }
}

public void mouseMoved ( MouseEvent mouseEvent )
///////////////////////////////////////////////////////////////////
{
    Point mousePoint = mouseEvent.getPoint ( );

    moveRequest.setDestination ( mousePoint.x, mousePoint.y );

    synchronizer.replace ( moveRequest );
}

public void mousePressed ( MouseEvent mouseEvent )
///////////////////////////////////////////////////////////////////
{
    fire ( );
}

///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////

private void fire ( )
///////////////////////////////////////////////////////////////////
{
    synchronizer.replace ( fireRequest );
}

```

键盘和鼠标事件会让 NetController 使用 Synchronizer 的 replace() 方法向游戏服务器发送消息。因为在最初的单机版 Mars 的开发过程中，控制器是从视图和模型中分离出来的，所以在多玩家联网版中用一个重定向到网络的控制器替换这个控制器就比较简单。

10.3.9 NetMain

就像这个游戏的单机版里的 Main 一样，类 NetMain 将模型、视图和控制器组合在一起。然而，在这种情况下，Synchronizer 是作为模型使用的，而 NetController 是作为控制器使用的。同时，视图代码则保持不变。

```
package com.croftsoft.apps.mars.net;

[...]
```

```
public final class NetMain
    extends AnimatedApplet
    //////////////////////////////////////
    //////////////////////////////////////
    {

    [...]

    private Synchronizer synchronizer;

    [...]

    public void init ( )
    //////////////////////////////////////
    {
        super.init ( );

        [...]

        // model

        String playerName
            = Long.toString ( new Random ( ).nextLong ( ) );
```

playerName 使用的是一个随机产生的字符串。在这里，它并不关心 playerName 到底是什么，只要对客户机而言是惟一的就可以了。

```
URL codeBaseURL = null;

try
{
    codeBaseURL = getCodeBase ( );
}
catch ( Exception ex )
{
}
```




```

try
{
    synchronizer
        = new Synchronizer ( playerName, codeBaseUrl );
    synchronizer.init ( );
}
catch ( Exception ex )
{
    ex.printStackTrace ( );
}

// view

GameAnimator gameAnimator = new GameAnimator (
    synchronizer,
    animatedComponent,
    getClass ( ).getClassLoader ( ),
    MEDIA_DIR,
    BACKGROUND_COLOR );

addComponentAnimator ( gameAnimator );

[...]

// controllers

[...]

new NetController ( playerName, synchronizer, animatedComponent );
}

public void start ( )
{
    ///////////////////////////////////////////////////
    synchronizer.start ( );

    super.start ( );
}

[...]

```

4 个生命周期方法都调用 `synchronizer` 实例上相应的方法。当游戏暂停和动画挂起的时候，联网线程将会挂起，游戏状态的本地副本也会变得过时。当游戏恢复的时候，再刷新本地副本。

10.4 可重用的服务器端代码

对这个示例游戏而言，可重用的服务器端代码包括一些像 `HttpGatewayServlet` 和 `Server` 这样的类，这些类在前面已经作过介绍。在服务器上使用 `XmlBeanCoder` 加载游戏初始化的值。`SerializableLib` 中的 `copy()` 方法用于创建服务器上游戏状态的快照。

10.4.1 XmlBeanCoder

在第 9 章中，StringCoder 用于对文本格式的网络消息进行编码和解析。com.croftsoft.core.beans 中的类 XmlBeanCoder 也可以用于实现这个目的，因为它可以对 XML 格式的消息进行编码和解析。然而，为了提高效率，在本章的这个游戏示例中，使用的是 SerializableCoder，因为它在编码的过程中还对消息进行压缩处理。

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
  <object class="com.croftsoft.apps.mars.net.GameInit">
    <void property="ammoDumpExplosion">
      <double>3.0</double>
    </void>
    [...]
    <void property="worldHeight">
      <int>400</int>
    </void>
    <void property="worldWidth">
      <int>600</int>
    </void>
  </object>
</java>
```

为了支持非 Java 客户端软件的实现，可以考虑为网络消息使用 XML 的 Encoder 和 Parser 实现，而不会去考虑文本可能会比压缩的串行化 Java 对象的效率更低。但是我不推荐使用 XmlBeanCoder 来完成这种任务，因为它产生的 XML 是专门为 Java bean 编码定制的一种格式。这可以从上面给出的对 GameInit 对象的编码输出中得到印证。本章的后面将介绍这个 GameInit 类。

如果客户端软件和服务器软件都是使用 Java 实现的，那么在开发和调试过程中，也许会临时使用 XmlBeanCoder，这样就可以使用一个网络流量分析器来监控人类可读格式的网络消息。如果您从来没有使用过这种网络流量分析器，就可能需要看一下 Ethereal，这个软件是和 Red Hat Linux 一起发布的。您可以获取 applet 和 servlet 之间的网络流量，并将其保存到一个文件中，以便以后进行检查。

```
package com.croftsoft.core.beans;

import java.beans.*;
import java.io.*;

import com.croftsoft.core.io.Encoder;
import com.croftsoft.core.io.Parser;

public final class XmlBeanCoder
  implements Encoder, Parser
  //////////////////////////////////////
  //////////////////////////////////////
{

  public static final XmlBeanCoder INSTANCE = new XmlBeanCoder ( );
```


当需要一个 Encoder 或 Parser 实例的时候，就可以使用一个单态的 INSTANCE。否则，就可以直接调用静态方法。

[...]

```
public static Object decodeFromXml ( InputStream inputStream )
////////////////////////////////////
{
    XMLDecoder xmlDecoder = new XMLDecoder ( inputStream );

    try
    {
        return xmlDecoder.readObject ( );
    }
    finally
    {
        xmlDecoder.close ( );
    }
}
```

[...]

尽管这里只给出了一个静态方法，但是 XmlBeanCoder 实际上提供了多个可重用的静态方法。这些静态方法都依赖于内核包 java.beans.XMLEncoder 中 XMLEncoder 和 XMLDecoder 的实例。在第 6 章“持久数据”一章中，介绍从 XML 文件中加载游戏初始化数据对象时，曾经讲过 XMLEncoder 和 XMLDecoder。

10.4.2 SerializableLib

在第 6 章中，已经介绍过 com.croftsoft.core.io 包中 SerializableLib 静态方法库类里的多个方法。本节主要介绍另一个新方法 copy()，这个方法的作用是创建 Serializable 对象的一个副本。

```
public static Serializable copy ( Serializable serializable )
    throws IOException
////////////////////////////////////
{
    ByteArrayOutputStream byteArrayOutputStream
        = new ByteArrayOutputStream ( );

    ObjectOutputStream objectOutputStream
        = new ObjectOutputStream ( byteArrayOutputStream );

    objectOutputStream.writeObject ( serializable );

    byte [ ] bytes = byteArrayOutputStream.toByteArray ( );

    ByteArrayInputStream byteArrayInputStream
        = new ByteArrayInputStream ( bytes );

    ObjectInputStream objectInputStream
        = new ObjectInputStream ( byteArrayInputStream );
```

```

try
{
    return ( Serializable ) objectInputStream.readObject ( );
}
catch ( ClassNotFoundException ex )
{
    throw new RuntimeException ( );
}
}

```

另一个更加高效的方法可能就是使用在类 `Object` 中声明的 `clone()` 方法。`clone()` 方法默认的实现是创建一个浅复制(shallow copy)，这种浅复制只创建根对象的一个副本。根对象引用的所有其他对象都通过赋值保存，并且不克隆它们自身。为了实现我们的目的，就需要一个深复制(deep copy)，在深复制中，会备份整个对象图中的所有对象。如果准备要使用克隆技术，那么就必须要在对象图(需要它确保执行的是深复制)的每一个类中重写这个 `clone()` 方法。

方法 `copy()` 总会产生一个深复制，即使根对象不是 `Cloneable` 的，只要它是 `Serializable` 的就可以。如果对象图中的某一个对象不是 `Serializable` 的，`copy()` 方法就会抛出 `IOException` 异常。一般情况下，我们都希望它在反串行化处理的过程中也能抛出一个 `ClassNotFoundException` 异常。但是，由于没有加载新类来创建这个副本，所以这个异常就被掩盖了。

10.5 游戏特定的服务器端代码

`MarsServlet` 将 HTTP 请求转发给 `MarsServer`。`MarsServer` 处理 `Player` 的修改请求，并查看 `NetGame` 的状态。`GameInit` 对象为服务器端的 `NetGame` 提供了初始值。

10.5.1 GameInit

`GameInit` 类提供游戏的一些初始化值，例如游戏世界的大小以及其中弹药库和障碍物的数量等。

```

package com.croftsoft.apps.mars.net;

[...]

public final class GameInit
    implements GameInitAccessor, Serializable, Testable
{
    ///////////////////////////////////////////////////
    ///////////////////////////////////////////////////

```

`GameInit` 类实现了 `accessor` 接口 `GameInitAccessor`，这个接口抽象游戏中具体类的实现，这样替换持久性机制就可以变得很容易。`GameInit` 类是专为使用对象串行化或 XML bean 编码的持久性机制而设计的。

```

[...]

public static GameInit createDefaultGameInit ( )
    ///////////////////////////////////////////////////

```



```

{
    GameInit gameInit = new GameInit ( );

    gameInit.setTimeFactorDefault ( DEFAULT_TIME_FACTOR );

    [...]

    gameInit.setAmmoDumpZ ( DEFAULT_AMMO_DUMP_Z );

    return gameInit;
}

```

静态方法 `createDefaultGameInit()` 创建一个带有默认值的 `GameInit` 类的实例。

```

public static void createTemplateXmlFile ( String filename )
    throws IOException
    //////////////////////////////////////
{
    XmlCoder.saveToXmlFile ( createDefaultGameInit ( ), filename );
}

```

静态方法 `createTemplateXmlFile()` 创建一个模板初始化文件。在本章前面的 `XmlBeanCoder` 这一节，已经给出了由这个方法所创建文件的一部分。这种人类可读的文件可以让游戏的设计者使用文本编辑器手动地进行修改，然后再以资源文件的形式存储在 WAR 文件内。它也可以在初始化以后由用户进行修改。应用程序服务器可以从硬盘上加载一个 Web 应用程序，包括所有修改后的资源文件，无论这些资源文件是否仍然是以原始压缩 WAR 包的形式存在的，还是以已经解压到部署目录的子目录中的形式存在的。例如，在部署过程中，用户可能会解压 WAR 文件，然后再编辑 HTML 文件以将 applet 的大小由 600×400 改变为 800×600 。为了让可视游戏世界的这个增大的部分对通过游戏服务器逻辑的运行来说有效，用户会接着修改 `/WEB-INF/mars.xml` 游戏初始化资源文件内的属性 `worldWidth` 和 `worldHeight`。

提示：

我喜欢将 applet 的默认大小定为 600×400 (宽和高，单位是像素)，因为它似乎正好适合在常规屏幕分辨率设置下的浏览器显示区域内进行显示，而且不需要带滚动条，或者只带垂直滚动条就可以了。我还喜欢使用 40 的倍数作为瓦片和精灵图片的大小，例如 40×40 ，或 80×40 ，这是因为 40 是很多标准显示区域大小 (例如 320×240 、 600×400 、 640×480 和 800×600) 最常用的一个整数因子。

```

public GameInit ( )
    //////////////////////////////////////
{
}

```

不带参数的构造函数将所有实例变量的值都初始化为 `null` 或 `0`。我们记得，在 XML Java bean 编码中，为了减少输出空间的大小，所有与通过不带参数的构造函数创建的实例变量的值相同的这些实例变量的值，在编码过程中都不写出。由于 `createDefaultGameInit()` 提供的默认值与由不带参数的构造函数初始化的这些值完全不同，所以所有的数据字段都会被包含在由 `createTemplateXmlFile()` 创建的这个临时文件中。

```
[...]

public double getTimeFactorDefault ( ) { return timeFactorDefault; }

[...]

public void setTimeFactorDefault ( double timeFactorDefault )
///////////////////////////////////////////////////////////////////
{
    this.timeFactorDefault = timeFactorDefault;
}
```

有很多这样的 **accessor** 方法和 **mutator** 方法，每一个参数都有这样的一对方法。这里仅仅给出了其中的一对。

10.5.2 Player

Player 对象是在服务器上进行维护的，这样来自于客户机的请求就可以持续地与虚拟游戏世界里具体的 **Tank** 模型相互联系在一起。

```
package com.croftsoft.apps.mars.net;

import com.croftsoft.core.lang.NullArgumentException;

import com.croftsoft.apps.mars.model.seri.SeriTank;

public final class Player
///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////
{

    private final String name;

    private final SeriTank seriTank;

    //

    private GameData gameData;

    private long      lastRequestTime;

    ///////////////////////////////////////////////////////////////////
    ///////////////////////////////////////////////////////////////////

    public Player (
        String    name,
        SeriTank seriTank )
    ///////////////////////////////////////////////////////////////////
    {
        NullArgumentException.check ( this.name    = name );
```



```

    NullPointerException.check ( this.seriTank = seriTank );
}

//////////////////////////////////////
// accessor methods
//////////////////////////////////////

public synchronized long getLastRequestTime ( )
//////////////////////////////////////
{
    return lastRequestTime;
}

public String    getName      ( ) { return name;      }

public SeriTank  getSeriTank ( ) { return seriTank; }

public GameData  getGameData ( ) { return gameData; }

//////////////////////////////////////
// mutator methods
//////////////////////////////////////
public void setGameData ( GameData gameData )
//////////////////////////////////////
{
    this.gameData = gameData;
}

public synchronized void setLastRequestTime ( long lastRequestTime )
//////////////////////////////////////
{
    this.lastRequestTime = lastRequestTime;
}

```

在每一个客户机请求中都会传输 Player 的 name 值。这个类确定了哪一个 seriTank 归 Player 所有和由哪一个 Player 进行控制。gameData 是这个 Player 可以访问的游戏状态的一个快照。lastRequestTime 用来保存 Player 最后和服务端进行联系的时间。由于 64 位 long 型值的赋值操作不能保证赋值的原子性，所以实例变量 lastRequestTime 的 accessor 方法和 mutator 方法是同步的。

10.5.3 NetGame

类 NetGame 封装了游戏的状态和逻辑。在 Mars 这个游戏的单机版本中相对应的类是 SeriGame。

```

package com.croftsoft.apps.mars.net;

[...]

public final class NetGame

```

```

////////////////////////////////////
////////////////////////////////////
{

```

```
private final SeriWorld seriWorld;
```

```
[...]
```

```
private final Queue      newPlayerNameQueue;
```

```
private final Map        nameToPlayerMap;
```

`newPlayerNameQueue` 存储在服务器上创建一个新 `Player` 对象的请求。`nameToPlayerMap` 存储 `Player` 的数据。

```
private final double     timeDeltaMax;
```

```
[...]
```

```
private final double     obstacleRadiusMin;
```

`NetGame` 使用了 `GameInit` 中的大部分值。那些被重复使用的值是以 `final` 实例变量的形式存储的。

```
private GameData gameData;
```

```
private SeriWorld copySeriWorld;
```

变量 `gameData` 和 `copySeriWorld` 提供了游戏状态最近更新后的快照备份。

```

public static NetGame load (
    GameInitAccessor gameInitAccessor,
    String            primaryFilename,
    String            backupFilename )
    throws ClassNotFoundException, IOException
////////////////////////////////////
{
    SeriWorld seriWorld = ( SeriWorld )
        SerializableLib.load ( primaryFilename, backupFilename );

    return new NetGame ( gameInitAccessor, seriWorld );
}

```

这里提供了静态方法 `load()`，目的是隐藏通过相应的 `save()` 方法将 `NetGame` 的实例保存到硬盘的具体细节。在这种情况下，`load()` 就会检索以压缩串行化对象的形式保存的 `SeriWorld` 实例。

```

public NetGame ( GameInitAccessor gameInitAccessor )
////////////////////////////////////
{
    this ( gameInitAccessor, ( SeriWorld ) null );
}

```



```

public NetGame ( )
///////////////////////////////////////////////////
{
    this ( ( GameInitAccessor ) null );
}

```

公有的构造函数委托给主构造函数。这个主构造函数被声明为私有的，这样就会隐藏 **SeriWorld** 类的使用(即具体实现的细节)。

```

public GameData getGameData ( )
///////////////////////////////////////////////////
{
    return gameData;
}

```

这个方法为旁观而没有参与的客户机检索当前游戏状态的快照。注意 **gameData** 没有包含具体玩家的坦克数据。

```

public Player getPlayer ( String playerName )
///////////////////////////////////////////////////
{
    Player player = ( Player ) nameToPlayerMap.get ( playerName );

    if ( player == null )
    {
        newPlayerNameQueue.replace ( playerName );
    }

    return player;
}

```

如果 **playerName** 没有被识别，创建一个新 **Player** 对象的请求就会在队列中排队，等待在稍后游戏循环的更新阶段进行的处理。

```

public void save (
    String primaryFilename,
    String backupFilename )
    throws IOException
///////////////////////////////////////////////////
{
    SerializableLib.save ( seriWorld, primaryFilename, backupFilename );
}

```

注意，只有 **seriWorld** 是持久的。**Player** 数据并没有保存到磁盘。

```

public void update ( )
///////////////////////////////////////////////////
{
    seriWorld.prepare ( );

    timekeeper.update ( );
}

```

```

double timeDelta = timekeeper.getTimeDelta ( );

if ( timeDelta > timeDeltaMax )
{
    timeDelta = timeDeltaMax;
}

seriWorld.update ( timeDelta );

```

除了更新虚拟游戏世界的状态和其中的全部模型以外，`update()`方法还执行很多修改游戏状态的其他任务，包括修改 `Player` 数据。

```

// restore and relocate destroyed obstacles

Obstacle [ ] obstacles = seriWorld.getObstacles ( );

for ( int i = 0; i < obstacles.length; i++ )
{
    Obstacle obstacle = obstacles [ i ];

    if ( !obstacle.isActive ( ) )
    {
        resetObstacle ( obstacle );
    }
}

```

部分游戏逻辑是所有被摧毁的障碍物都立即重置。这会在新的位置上重新显示它们。

```

// create list of owned tanks and disconnected players

java.util.List tankList = new ArrayList ( );

java.util.List removeList = new ArrayList ( );

Iterator iterator = nameToPlayerMap.values ( ).iterator ( );

long currentTimeMillis = System.currentTimeMillis ( );

while ( iterator.hasNext ( ) )
{
    Player player = ( Player ) iterator.next ( );

    tankList.add ( player.getSeriTank ( ) );

    long lastRequestTime = player.getLastRequestTime ( );

    if ( currentTimeMillis >= lastRequestTime + playerTimeout )
    {
        removeList.add ( player );
    }
}

```


该代码创建了游戏世界中所有仍然与 Player 对象关联的坦克的一个列表，以及最近没有再与服务器进行通信的那些玩家的另一个列表。

```
// remove disconnected players and their tanks

iterator = removeList.iterator ( );

while ( iterator.hasNext ( ) )
{
    Player player = ( Player ) iterator.next ( );

    seriWorld.remove ( player.getSeriTank ( ) );

    nameToPlayerMap.remove ( player.getName ( ) );
}
```

通过 playerTimeout 期限进行确定，如果一个 player 最近没有产生新的请求，就会认为游戏已经将这个 player 删除，并且将该 player 的坦克从游戏世界中删除。

```
// remove unowned tanks and reactivate dead player tanks

Tank [ ] tanks = seriWorld.getTanks ( );

for ( int i = 0; i < tanks.length; i++ )
{

    Tank tank = tanks [ i ];

    if ( !tankList.contains ( tank ) )
    {
        seriWorld.remove ( tank );
    }
    else if ( !tank.isActive ( ) )
    {
        for ( int j = 0; j < attemptsMax; j++ )
        {
            tank.initialize (
                worldWidth * actionsRandom.nextDouble ( ),
                worldHeight * actionsRandom.nextDouble ( ) );

            if ( !seriWorld.isBlocked ( tank ) )
            {
                break;
            }
        }
    }
}
```

如果坦克不再与 Player 关联在一起，就会将它从游戏世界中删除。如果玩家的坦克还没有被摧毁，它就会在一个新位置上恢复，这样战斗就可以重新继续。

```
// create new players and their tanks

while ( true )
```

```

{
    String newPlayerName = ( String ) newPlayerNameQueue.poll ( );

    if ( newPlayerName == null )
    {
        break;
    }

    Player player = getPlayer ( newPlayerName );

    if ( player != null )
    {
        continue;
    }

    Tank playerTank = seriWorld.createTank (
        0.0, 0.0,
        new Color (
            actionsRandom.nextInt ( 256 ),
            actionsRandom.nextInt ( 256 ),
            actionsRandom.nextInt ( 256 ) ) );

    [...]

    playerTank.setTankOperator (
        new DirectedTankOperator ( playerTank ) );

    player = new Player ( newPlayerName, ( SeriTank ) playerTank );

    nameToPlayerMap.put ( newPlayerName, player );
}

```

这段代码创建了一个新 **Player** 对象，并将它添加到游戏中。它的 **playerTank** 被添加到游戏世界中。由于玩家的数据被认为是游戏状态的一部分，所以游戏对象的所有操作都必须在游戏循环线程的更新阶段中执行。

```

// update snapshots

try
{
    copySeriWorld = ( SeriWorld ) SerializableLib.copy ( seriWorld );
}
catch ( IOException ex )
{
    // This normally will never happen.

    throw ( RuntimeException )
        new RuntimeException ( ).initCause ( ex );
}

gameData = new GameData ( copySeriWorld, null );

```



```

iterator = nameToPlayerMap.values ( ).iterator ( );

while ( iterator.hasNext ( ) )
{
    Player player = ( Player ) iterator.next ( );
    try
    {
        SeriTank copySeriTank = ( SeriTank )
            SerializableLib.copy ( player.getSeriTank ( ) );

        player.setGameData (
            new GameData ( copySeriWorld, copySeriTank ) );
    }
    catch ( IOException ex )
    {
        // This normally will never happen.

        throw ( RuntimeException )
            new RuntimeException ( ).initCause ( ex );
    }
}
}

```

在更新阶段的最后，创建游戏状态的新快照。在更新阶段正在处理的过程中，所有查看游戏状态的请求都会接收到前面保存的快照。在更新阶段的最后，使用对象赋值的方法，将原来的游戏状态快照替换为新的游戏状态快照，这样保证了赋值的原子性，因此也是线程安全的。注意，每一个 Player 对象都使用它自己的 GameData 快照进行更新，这个 GameData 也包括玩家特定的坦克信息。

```

private NetGame (
    GameInitAccessor gameInitAccessor,
    SeriWorld seriWorld )
////////////////////////////////////
{
    if ( gameInitAccessor == null )
    {
        gameInitAccessor = GameInit.createDefaultGameInit ( );
    }

    [...]

    if ( seriWorld == null )
    {
        [...]
    }
    else
    {
        this.seriWorld = seriWorld;
    }

    [...]
}

```

```

newPlayerNameQueue = new ListQueue ( );

nameToPlayerMap = new HashMap ( );

try
{
    copySeriWorld = ( SeriWorld ) SerializableLib.copy ( seriWorld );
}
catch ( IOException ex )
{
    // This normally will never happen.

    throw ( RuntimeException )
        new RuntimeException ( ).initCause ( ex );
}

gameData = new GameData ( copySeriWorld, null );
}

[...]
```

如果构造函数的参数为空，私有的构造函数就提供默认的 `gameInitAccessor` 和 `seriWorld` 的实例。游戏状态的快照也会被初始化。

10.5.4 MarsServer

`MarsServer` 负责处理玩家的请求。它还管理游戏循环线程，这个线程周期性地更新游戏的状态。

```

package com.croftsoft.apps.mars.net;

[...]
```

```

public final class MarsServer
    implements Commissionable, Server
{
    //////////////////////////////////////
    //////////////////////////////////////
    {
```

类 `MarsServer` 实现了 `com.croftsoft.core.lang.lifecycle` 包中的接口 `Commissionable`，以表示它提供了一个 `init()` 和一个 `destroy()` 方法。当有一组需要在启动的时候初始化和在关闭的时候销毁的 `Commissionable` 对象时，这个接口非常有用。它也实现了接口 `Server`，这样它就可以通过 `HttpGatewayServlet` 子类，以通用的形式使用。

```

private static final boolean DEBUG          = true;

private static final double UPDATE_RATE     = 30.0;

private static final long SAMPLE_PERIOD     = 10 * 1000;

private static final long REQUEST_TIMEOUT  = 10 * 1000;
```


DEBUG 为 true 时, 网络请求速度和游戏循环线程的状态就会输出到控制台。每秒 30 次的 UPDATE_RATE 用来控制游戏循环的更新频率。计算网络请求频率的采样周期 SAMPLE_PERIOD 是 10 秒。如果在 10 秒的 REQUEST_TIMEOUT 的限制范围内没有玩家的请求, 游戏循环将会挂起, 直到有下一个玩家请求。

```
private final String primaryFilename;

private final String fallbackFilename;

private final Looper looper;

//

private GameInitAccessor gameInitAccessor;

private NetGame          netGame;

private long              count;

private long              startTime;

private long              lastRequestTime;
```

primaryFilename 和 fallbackFilename 用来加载一个前面保存的 netGame。Looper 驱动服务器端的游戏循环。count 是自上一个 startTime 以后已经得到服务的那些请求的一个计数。

```
public MarsServer (
    String primaryFilename,
    String fallbackFilename )
    //////////////////////////////////////
{
    this.primaryFilename = primaryFilename;

    this.fallbackFilename = fallbackFilename;

    setGameInitAccessor ( GameInit.createDefaultGameInit ( ) );

    looper = new Looper (
        new Loopable ( )
        {
            public boolean loop ( )
            {
                return MarsServer.this.loop ( );
            }
        },
        new FixedDelayLoopGovernor ( UPDATE_RATE ),
        null,
        ( String ) null,
        Thread.MIN_PRIORITY,
        true );
}
```

```

////////////////////////////////////
////////////////////////////////////

public void setGameInitAccessor (
    GameInitAccessor gameInitAccessor )
    //////////////////////////////////
{
    NullArgumentException.check (
        this.gameInitAccessor = gameInitAccessor );
}

```

默认的 `gameInitAccessor` 是在构造函数内部设置的。这里提供了这个 `mutator` 方法 `setGameInitAccessor()`，这样它就可以由从初始化文件中加载的一个实例所替换。如果要使用它，就在调用 `init()` 方法以前调用它。

```

public void init ( )
    //////////////////////////////////
{
    if ( primaryFilename != null )
    {
        try
        {
            netGame = NetGame.load (
                gameInitAccessor, primaryFilename, fallbackFilename );
        }
        catch ( FileNotFoundException ex )
        {
        }
        catch ( Exception ex )
        {
            ex.printStackTrace ( );
        }
    }

    if ( netGame == null )
    {
        netGame = new NetGame ( gameInitAccessor );
    }
}

```

如果不能从保存的游戏文件中加载 `netGame`，就会使用 `gameInitAccessor` 提供的初始值创建一个新的实例。

```

startTime = System.currentTimeMillis ( );

lastRequestTime = startTime;

looper.init ( );
}

```

即使它是在这里进行初始化的，但是直到接收到第一个请求以前，`looper` 也不会真正地启动起来。


```

public Object serve ( Object requestObject )
///////////////////////////////////////////////////
{
    synchronized ( this )
    {
        lastRequestTime = System.currentTimeMillis ( );

        if ( DEBUG )
        {
            ++count;
        }
    }
}

```

每次调用 `serve()` 方法的时候，都会更新 `lastRequestTime`，同时也会递增 `count`。与其他原子类型和对象的赋值不同，64 位 `Long` 型值的赋值不能保证赋值操作的原子性。如果在进行赋值的同时，另一个线程试图读取这个值，那么，就可能导致一些问题。为此，`lastRequestTime` 和 `count` 赋值就会在一个同步块中执行。

```

if ( !( requestObject instanceof Request ) )
{
    throw new IllegalArgumentException ( );
}

looper.start ( );

```

如果游戏循环被挂起，这就会重新将它启动。如果没有被挂起，在 `looper` 上调用 `start()` 方法就没有任何作用。

```

Request request = ( Request ) requestObject;

String playerName = request.getPlayerName ( );

```

`playerName` 是从请求中检索出来的。这是可能要运行使用密码或数字签名检查认证的代码的地方。

```

if ( playerName == null )
{
    return netGame.getGameData ( );
}

```

如果 `playerName` 为空，这个方法就会返回游戏状态的一个快照，而不会返回任何坦克的数据。这是由正在显示游戏而没有参与游戏的客户机使用的。

```

Player player = netGame.getPlayer ( playerName );

if ( player == null )
{
    return netGame.getGameData ( );
}

player.setLastRequestTime ( System.currentTimeMillis ( ) );

```

除了更新服务器上一次从某个客户机那里接收一个连接的时间作为 `lastRequestTime` 以外，某个具体玩家上一次作出一个请求的时间也会在 `player` 对象内记录下来。

```
if ( request instanceof ViewRequest )
{
    GameData gameData = player.getGameData ( );

    if ( gameData == null )
    {
        gameData = netGame.getGameData ( );
    }

    return gameData;
}
```

返回玩家特定视图的 `gameData`，这个 `gameData` 中包含玩家的坦克数据。如果刚刚创建 `player`，那么 `gameData` 就可能会为 `null`。在这种情况下，就会返回非玩家特定的 `gameData`。

```
TankOperator tankOperator
    = player.getSeritank ( ).getTankOperator ( );

if ( request instanceof MoveRequest )
{
    MoveRequest moveRequest = ( MoveRequest ) request;

    tankOperator.go ( moveRequest.getDestination ( ) );

    return null;
}

if ( request instanceof FireRequest )
{
    tankOperator.fire ( );

    return null;
}

throw new IllegalArgumentException ( );
}
```

`MoveRequest` 或 `FireRequest` 被适当转换为玩家坦克 `tankOperator` 上方法的调用。任何其他类型的请求都将会产生一个 `IllegalArgumentException` 异常。注意，`tankOperator` 的方法 `go()` 和 `fire()` 并没有立刻为这些请求提供服务。相反，它们只是简单地存储这些请求，直到可以在游戏循环的更新阶段执行它们为止。这是线程安全的，因为目标对象的赋值是原子性的。如果有多个请求到达的速度比能够处理的速度要快，那么最新到达的请求就会替换以前的所有请求。例如，如果在更新之间，对同一个 `tankOperator` 两次调用了方法 `go()`，那么第二次的目标就会覆盖第一次的目标。

```
public void destroy ( )
{
    //////////////////////////////////////
```



```

    looper.stop ( );

    looper.destroy ( );

    try
    {
        synchronized ( this )
        {
            netGame.save ( primaryFilename, fallbackFilename );
        }
    }
    catch ( Exception ex )
    {
        ex.printStackTrace ( );
    }
}

```

`destroy()`方法会挂起和销毁主游戏循环。如果循环已经挂起，调用 `looper` 的 `stop()`方法将不会有任何效果。在服务器关闭的时候，`netGame` 会保存到磁盘。在重新启动服务器的时候，障碍物和临时弹药库就会恢复到它们以前的位置和状态。

我们知道，在完成当前的循环迭代以后，`destroy()`方法并没有立即终止游戏循环，而是发出游戏循环应该终止的信号。为此，`netGame` 对 `save()`方法的调用必须是同步的，这样它才不会覆盖上一次游戏循环的更新阶段。

```

private synchronized boolean loop ( )
///////////////////////////////////////////////////////////////////
{
    netGame.update ( );
    long currentTime = System.currentTimeMillis ( );

    if ( DEBUG )
    {
        if ( currentTime >= startTime + SAMPLE_PERIOD )
        {
            System.out.println ( "requests per second: "
                + ( MathConstants.MILLISECONDS_PER_SECOND * count )
                / ( currentTime - startTime ) );

            startTime = currentTime;

            count = 0;
        }
    }

    if ( currentTime >= lastRequestTime + REQUEST_TIMEOUT )
    {
        if ( DEBUG )
        {
            System.out.println ( "MarsServer game loop pausing..." );
        }
    }
}

```

```

        return false;
    }

    return true;
}

```

私有方法 `loop()` 更新了 `netGame`。如果设置了 `DEBUG`，它还输出请求的统计数据。如果最近没有玩家的请求，那么 `loop()` 方法就必须返回 `false`，表示主循环应该挂起，以便于那些还没有使用的处理时间对服务器上运行的其他程序可用。`loop()` 方法被同步，这样它就不会在并发的用户请求线程进行更新的时候，试图读取 `lastRequestTime` 和请求计数 `count`，并且在关闭的过程中，也不会和游戏状态保存到磁盘的时候试图更新。

10.5.5 MarsServlet

类 `MarsServlet` 扩展了 `HttpGatewayServlet`，以便为超类提供 `Server` 接口实现 `MarsServer`。它还通过由 `servlet` 上下文可以访问的资源文件加载游戏初始化数据。

```

package com.croftsoft.apps.mars.net;

[...]

public final class MarsServlet
    extends HttpGatewayServlet
    //////////////////////////////////////
    //////////////////////////////////////
    {

    [...]

    private static final String GAME_INIT_PATH = "/WEB-INF/mars.xml";

    private static final String PRIMARY_FILENAME = "mars.dat";

    private static final String FALLBACK_FILENAME = "mars.bak";

    //

    private final MarsServer marsServer;

    //////////////////////////////////////
    //////////////////////////////////////

    public MarsServlet ( )
    //////////////////////////////////////
    {
        this ( new MarsServer ( PRIMARY_FILENAME, FALLBACK_FILENAME ) );
    }

    [...]

```


不带参数的 MarsServlet 构造函数将一个新创建的 MarsServer 实例传递给私有的主构造函数。

```
public void init ( )
    throws ServletException
    //////////////////////////////////////
{
    System.out.println ( SERVLET_INFO );

    try
    {
        GameInit gameInit = ( GameInit ) XmlBeanCoder.decodeFromXml (
            getServletContext ( ).getResourceAsStream ( GAME_INIT_PATH ) );

        marsServer.setGameInitAccessor ( gameInit );

        marsServer.init ( );
    }
    catch ( Exception ex )
    {
        throw ( ServletException )
            new UnavailableException ( ex.getMessage ( ) ).initCause ( ex );
    }
}

public void destroy ( )
    //////////////////////////////////////
{
    try
    {
        marsServer.destroy ( );
    }
    catch ( Exception ex )
    {
        log ( ex.getMessage ( ), ex );
    }
}
```

MarsServlet 保存指向 marsServer 的一个引用，因此只要初始化或销毁 servlet，就可以也将 MarsServlet 初始化或销毁。在第一次访问 servlet 的时候，servlet 一般都会被 servlet 容器初始化，当关闭 servlet 容器的时候，servlet 一般也都会被销毁。您可能认为 marsServer 的 init() 方法能打开一个 JDBC 连接，而相应的 destroy() 方法可能会关闭这个连接。另一个可能性就是 init() 方法使用 Java 命名和目录接口(Java Naming and Directory Interface, JNDI)为游戏服务器定位一个 EJB 会话或实体 bean。或者可能是这样的，即 init() 方法只是在当前虚拟机内的内存中创建游戏的一个新实例，就像这里的实现一样。

```
private MarsServlet ( MarsServer marsServer )
    //////////////////////////////////////
{
    super (
        marsServer,
```

```
        SerializableCoder.INSTANCE,  
        SerializableCoder.INSTANCE );  
  
        NullArgumentException.check ( this.marsServer = marsServer );  
    }
```

MarsServlet 的私有主构造函数委托给 HttpGatewayServlet 超类的构造函数。MarsServlet 子类构造函数也保存指向 MarsServer 的一个实例变量的引用，这样就可以从这个类访问它。

10.6 小结

本章介绍了多玩家联网游戏的代码，这个多玩家联网游戏是从一个单机版游戏转换来的。这是通过 MVC 实现的。服务器和客户机之间的网络同步是使用整个游戏状态的简单 HTTP 轮询维护的。来自于多个玩家客户机的修改游戏状态的并发请求，包括玩家数据，都被存储或放在队列中，以便稍后在服务器端游戏循环线程的更新阶段中进行串行地处理。查看游戏状态的并发请求，是通过返回存储的快照备份(这种快照备份是在每一个游戏循环的末尾，以一种线程安全的方式更新的)得到及时服务的。本章介绍了很多可重用类和游戏特定类。下一章还会再讨论这些类，因为下一章将介绍多玩家联网游戏的另一种状态同步机制。

10.7 参考文献

Feldman, Ari. "Designing for Different Display Modes." Chapter 2 in *Designing Arcade Computer Game Graphics*. Plano, TX: Wordware Publishing, 2001.



第 11 章 HTTP Pulling 机制

哪怕是最好的话题也不能讨论得过多。

——本杰明·富兰克林

在前一章中，我们使用 HTTP 轮询机制在客户机和服务器之间进行同步。这种方法的主要缺点是，即使游戏的状态没有更新，轮询也需要连续地将周期性的请求从客户机发送到服务器。另外游戏状态快照的周期性轮询还可能会错过在两次采样之间发生的一些事件。在前一章测试网络游戏的示例中，您可能已经注意到了这一点：那些只持续一帧的爆炸有时会在客户端视图的动画中被跳过。

理想的情况下，您会更愿意使用事件驱动的联网技术代替轮询机制。服务器上的游戏状态进行很频繁地更新时，这种机制会将网络流量降到最小。这种方法也能够保证即使是瞬间的状态变更也会被传递到客户端。一般来说，只要发生了对服务器端游戏状态的更新，事件驱动的通信就需要游戏服务器和玩家的客户端进行联系。当有网络防火墙和无符号的 applet 安全沙箱限制，阻止服务器与客户机进行通信时，这该怎样实现呢？

在第 9 章介绍的接口 Queue 中，我曾指出 pull() 方法会锁住调用线程，直到有一个从队列中检索出来的可用对象为止，从而将术语 pull 和 poll 进行区分。在 HTTP Pulling 中，客户端会建立与服务器的通信，就像 HTTP 轮询一样，但是，前者不是下载游戏状态的快照，而是客户端在服务器端消息队列上拖拉。如果队列中包含有一个事件，服务器就会立即进行响应。如果没有，服务器将会延迟它对客户端的响应直到有一个事件发生为止。当服务器上虚拟游戏世界中没有发生任何事件的时候，客户端和服务端之间的通信流量就会接近于完全停止。只要发生了事件，服务器就会立即进行响应，模拟服务器到客户端的事件通知的效果而不会违反其安全性。

11.1 测试示例

本章中的这个示例是 applet 和 servlet 的一个混合物，被称为 Chat。它将文本消息传递与动画混合在一起。当用户输入文本消息的时候，该文本消息就出现在所有已经与服务器建立连接的其他客户机的文本区域内。另外，当用户单击动画区域时，代表用户的一个虚拟人物——化身(avatar)就会穿越屏幕。这种穿越屏幕的移动过程会同时映射到所有的客户机上。如果虚拟空间里的两个化身之间发生了碰撞，就会锁定这种移动。用户也可以使用一个下拉菜单项目列表来更改他们化身的图像。由于使用的是 HTTP Pulling，所以当发生这种变更的时候，所有和服务器建立连接的客户机也都会立即更新。

```
appletviewer http://localhost:8080/chat/
```


这个示例使用的 Ant 构建目标是 chat_install。上面给出了默认的 URL。可以通过同时使用两个或多个 applet 客户机测试这个事件驱动的同步机制,就像在图 11-1 中列出的一样。为了避免造成迷惑,可以改变每一个客户机的化身图像,以使它们更加容易进行区别。



图 11-1 事件驱动的联网技术

只要 servlet 接收到一个来自于客户机的请求和只要 applet 接收到来自于服务器的一个事件,调试信息就会显示在 servlet 和 applet 的控制台窗口上。注意,只要输入了一个文本消息或由鼠标单击触发了化身的移动,服务器首先就会接收到一个来自于客户机的请求,然后所有客户机都会接收到来自服务器的一个广播事件。

还要注意,当化身到达它目标的时候,它移动的结束并不是由一个事件表示的,因为这个结果可以被客户端预知。在接收到一个移动开始事件的时候,客户端视图会以一个固定的速度,将它们化身模型的本地副本滑向目标点。一旦它们到达了虚拟空间的目标,这个化身就会自动停止。

仅当因为另一个化身的非期望阻塞而导致移动的过早结束时,才准许有来自于服务器的一个报告事件。由于网络滞后的原因,在发生碰撞的时候,有些化身看起来好像是“反弹”了一下。发生这种现象是因为客户端视图会将化身相互滑过对方,直到它们接收到来自于服务器的阻塞事件消息。这强制客户端视图将模型沿着计划好的路径向后移动一步或两步。

在这个示例中,每次创建客户端的时候,都会产生一个新的随机用户的名称,这是为了进行演示而专门设计的。当它第一次和服务器建立连接的时候,就会在虚拟世界里创建一个相应的化身,这个化身和那个用户名称关联在一起。如果客户端和服务器失去联系的时间超过一分钟,这个化身就会被删除,它们之间的关联也会被销毁。可以通过产生一个客户端视图并立即关闭这个视图的方法,观察这种现象。当第二次用一个新的随机分配的用户名称重新启动客户机的时候,就会有化身。如果这两个化身重叠在一起,可能就需要移动其中的一个化身,以观察这个现象。大约一分钟以后,第一个化身就会消失。

服务器上的游戏状态和客户端副本之间的同步是使用更新事件进行管理的。这些事件仅包含足够表示自上一次更新事件以后所发生变更的信息。如果客户端没有收到更新事件消息，或如果对服务器进行了重新设置，那么同步可能会失败。在这两种情况下，客户端都必须检测同步的失败并请求对玩家可见的整个游戏状态的完全更新。

可以通过在客户端视图仍然是活动的时候就关闭和重启服务器，来测试在这个示例中使用的客户机-服务器游戏状态再同步算法的健壮性。当客户机第一次检测到服务器没有响应的时候，它就会使用自适应的轮询逐渐地增加它试图连接之间的时间间隔。一秒或两秒以后，当服务器恢复的时候，客户机就会再次成功地和服务器建立联系，并检测服务器游戏状态的变更。由于在这个示例中，服务器端的游戏状态并不是持久的，所以客户端会需要请求一个新的化身。

也可以通过将客户端视图暂停一分钟以上的时间，来测试这个再同步算法。将客户端窗口最小化会导致动画和联网线程挂起。如果再很快地将其最大化，那些已经错过的事件都会被接收到，客户端视图也会重新跟上。但是如果客户端网络挂起的时间太长，在服务器上为客户端保存的事件就会丢失。客户端会自动地检测这个条件，并请求一个全屏刷新。

如果一个单独的更新事件丢失了(尽管这不容易被测试)，客户端也会请求一个全屏刷新。这种丢失是由客户端代码测试的，因为服务器是连续地为事件进行编号的。由于在虚拟的环境中，分隔开的客户端视图可能会收到基于它们各自化身视图的位置和区域的不同事件，所以每一个客户端都有它自己惟一的事件编号序列，这种编号序列的开头是一个随机产生的 64 位的数字。开头采用随机数字是为了降低第一个事件被错误地作为已经接收过的事件而丢弃的可能性。

11.2 可重用的客户端代码

Authentication 对象保存用户的名称和密码。Id 为远程对象提供了一个持久的引用。LongId 是具有惟一性的、长度为 64 位的 Id。ModelId 是化身的 Id。SeriModelId 是 ModelId 的一个实现。

11.2.1 Authentication

com.croftsoft.core.security 包中的类 Authentication 只是保存用户和密码的一对 String 值。它是 Serializable 的，所以可以通过网络进行发送，验证发送到服务器的所有客户机请求。记住，当通过网络发送密码的时候，您可能会使用 HTTPS 进行加密。这个示例并没有这样做。如果游戏的安全需求较低，也可以将密码设为空。

11.2.2 Id

com.croftsoft.core.util.id 包中的接口 Id 的实现是用来惟一地标识一个对象的。从某种意义上说，它好像是一个跨进程对象引用值。这个标识在时间和空间上都是一致的。在时间上，对象会以持久引用的形式维持它的 Id 值，即使对象被串行化到磁盘和在一个较晚的时间被恢复到内存的某一个新位置中。在空间上，一个分布式对象会由它的 Id 在网络上进行惟一标识。

实现了语义接口 Id 的对象是恒定不变的，同时也是 Serializable 和深 Cloneable 的。另外，equals() 和 hashCode() 方法的返回结果也不是基于瞬时数据的，而是在整个处理过程中具有一致性的。例如内核类 String 和 Long 的实例就拥有所有这些特征。

11.2.3 LongId

LongId 只是接口 Id 的一个实现，这个实现是由 Long 的一个实例支持的。

```
package com.croftsoft.core.util.id;

import com.croftsoft.core.lang.NullArgumentException;

public class LongId
    implements Id
    //////////////////////////////////////
    //////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

    //

    private final Long l;

    [...]

    public LongId ( Long l )
    //////////////////////////////////////
    {
        NullArgumentException.check ( this.l = l );
    }

    [...]

    public Object clone ( )
    //////////////////////////////////////
    {
        try
        {
            return super.clone ( );
        }
        catch ( CloneNotSupportedException ex )
        {
            // This will never happen.

            throw new RuntimeException ( );
        }
    }

    public boolean equals ( Object other )
    //////////////////////////////////////
    {
        if ( ( other == null )
            || !getClass ( ).equals ( other.getClass ( ) ) )
        {
            return false;
        }
    }
}
```



```

    return l.equals ( ( ( LongId ) other ).l );
}

public int hashCode ( )
///////////////////////////////////////////////////
{
    return l.hashCode ( );
}

[...]
```

注意，LongId 委托给一个 Long 对象实例变量 l，以便为 equals() 和 hashCode() 方法提供值。同一个包中的类 StringId 也是以类似的形式实现的，只是它委托给 String 的一个实例。

11.2.4 ModelId

com.croftsoft.core.animation.model 包中的接口 ModelId 是 Id 的一个扩展，它唯一地标识了一个 Model。这允许在服务器上的虚拟空间里模拟出来的一个实体可以与客户机上相应的副本进行配对。这个接口没有定义方法或常量。

11.2.5 SeriModelId

类 SeriModelId 是 LongId 的扩展，它实现了 ModelId。我更喜欢为我的 Id 实现使用 LongId 的子类，因为我喜欢为存储在关系数据库中的对象使用 integer 类型的键值。这样也很容易通过随机产生的一个 64 位的数字创建一个惟一的 LongId。

```

package com.croftsoft.core.animation.model.seri;

import com.croftsoft.core.util.id.LongId;

import com.croftsoft.core.animation.model.ModelId;

public final class SeriModelId
    extends LongId
    implements ModelId
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{

    private static final long serialVersionUID = 0L;

///////////////////////////////////////////////////
///////////////////////////////////////////////////

    public SeriModelId ( long l )
///////////////////////////////////////////////////
    {
        super ( l );
    }
}
```

您可能奇怪为什么使用 `ModelId` 的实现来标识远程的对象，而不是使用一个 `Long` 类型的值或一个更加简单的 `String` 类型的对象。其中的主要原因是这样的：这些持久且分布式的引用都有与常规引用相关的强类型方面的一些问题。

```
public void doSomething (
    String  userId,
    String  modelId );

public void doSomething (
    UserId  userId,
    ModelId modelId );
```

比较上面两个假定的方法签名。即使使用的是 `String` 类型的引用而不是指针，第一个方法签名也可能会导致一个问题——因为如果给错了参数顺序的话，代码仍然会成功通过编译。如果第二个方法(在这个方法中，每一个参数使用的都是 `Id` 的不同子接口)的参数顺序错了，在编译的时候就可以检测到错误。一般来说，只要需要标识可能存在于当前虚拟机器以外的对象，始终使用一个具体 `Id` 的子接口引用是一个很不错的主意。

11.3 游戏特定的客户端代码

客户端将 `Request` 在请求队列中排队，以便将其发送给游戏服务器。如果这个请求是 `CoalesceableRequest`，它可能用前面添加到队列里的请求替换这个请求。用户输入事件通过 `ChatController` 转换为适当的 `Request` 类型的一个实例。游戏服务器可能同步也可能异步地返回一个 `Response`。当客户端向服务器发送一个 `CreateUserRequest` 的时候，服务器就会向客户端返回一个 `CreateUserResponse`，并通过 `CreateUserConsumer` 在客户端进行处理。只要服务器端游戏状态被修改，就会产生一个 `Event`。`ChatClient` 提供了联网代码上传客户端 `Request` 的实例和下载服务器信息，例如 `Event` 和 `Response` 对象。`ChatSynchronizer` 使用服务器消息里的这些信息同步游戏状态的客户端副本。`ChatPanel` 提供了客户端视图，这使这些变更能够对玩家可见。`ChatApplet` 组合了分别作为模型、视图和控制器的 `ChatClient`、`ChatPanel` 和 `ChatController`。

11.3.1 Request

`com.croftsoft.apps.chat.request` 包中的接口 `Request` 扩展了 `Serializable`，因此它可以通过网络，从客户机上传到服务器。它定义了一个方法 `getAuthentication()`，这样请求的发起者就可以被服务器识别。接口 `Request` 的实现提供了其他一些方法。`AbstractRequest` 是 `Request` 的一个抽象类实现，它提供了一个默认的 `getAuthentication()` 的方法体实现。

11.3.2 CoalesceableRequest

`CoalesceableRequest` 是一个扩展了 `AbstractRequest` 的抽象类。它表示当前的请求可以用队列里的前一个请求进行替换。`CoalesceableRequest` 的子类包括 `CreateUserRequest`、`CreateModelRequest`、`MoveRequest`、`PullRequest`、`TalkRequest` 和 `ViewRequest`。`CreateUserRequest` 在服务器上创建一个玩家对象；`CreateModelRequest` 为玩家创建一个化身 `Model`；`MoveRequest`

移动或停止移动虚拟空间中的化身；PullRequest 从服务器上拖出一个消息；TalkRequest 发送文本消息；ViewRequest 获取服务器端游戏状态的一个完整快照。

```
public boolean equals ( Object other )
///////////////////////////////////////////////////
{
    if ( other == null )
    {
        return false;
    }

    return this.getClass ( ) == other.getClass ( );
}
```

重写 CoalesceableRequest 的这个 equals()方法，重新定义相等性比较的判定过程。如果 CoalesceableRequest 子类的两个实例都是同一个类的实例，那么就认为一个实例与另一个实例相等。这意味着一个 MoveRequest 的实例将会被认为等于另一个 MoveRequest 的实例。由于会认为它们是相等的，所以在使用 replace()方法添加到 Queue 中的时候，后到达的这个 MoveRequest 就会替换先达到的那个 MoveRequest。

注意 Queue 的 replace()方法替换它通过顺序搜索队列里的元素找到的第一个相等元素。这意味着当请求被更新的时候，不会失去它们在队列里的位置。如果一个 MoveRequest 在被一个新的、包含更新后目标的 MoveRequest 实例替换的时候，已在这一行的最前面，并准备上传到服务器，那么下次才上传新的 MoveRequest。

对 CoalesceableRequest 的大部分子类而言，将它们在队列内进行合并的原因显而易见。如果玩家在高速的替换中产生了多个 MoveRequest 或 ViewRequest 实例，例如，在这些请求被上传到服务器以前，那么丢弃除了最近的这个请求以外的所有请求就成了很有意义的事情。决定将 TalkRequest 更改为 CoalesceableRequest 的目的是防止玩家向服务器和其他玩家的客户端乱发消息。如果玩家不停地单击键盘并按 Enter 键就会发生这种事情。如果玩家产生 TalkRequest 实例的速度比它们可以通过网络进行发布的速度要快，那么它们就将会被合并，有些文本就会丢失。

11.3.3 ChatController

前面一章介绍的控制器示例 NetController 会将请求传递给联网对象，以便将其上传到服务器。在这一章中，这个 ChatController 将请求传递到队列中。因为使用的是队列而不是联网对象，所以 ChatController 就即可以用在单机模式中，也可以用在多玩家联网模式，而不需要进行任何修改。使用队列作为中间媒介降低了发送对象和接收对象之间的耦合。

```
package com.croftsoft.apps.chat.controller;

[...]

public final class ChatController
    extends UserInputAdapter
    implements ActionListener, ItemListener
```

```

////////////////////////////////////
////////////////////////////////////
{

[...]

public void mousePressed ( MouseEvent mouseEvent )
////////////////////////////////////
{
    if ( mouseEvent.getSource ( ) == moveComponent )
    {
        Point mousePoint = mouseEvent.getPoint ( );

        requestQueue.replace (
            new MoveRequest (
                authentication,
                new Point2DD ( mousePoint.x, mousePoint.y ) ) );
    }
    else
    {
        throw new IllegalArgumentException ( );
    }
}
}

```

在方法 `mousePressed()` 中, `MouseEvent` 被转换为 `MoveRequest`, 并被插入到 `requestQueue` 中。注意这里将请求添加到队列, 使用的是 `replace()` 方法而不是 `append()` 方法。

11.3.4 Response

`com.croftsoft.apps.chat.response` 包中的接口 `Response` 扩展了 `Serializable`, 这样它就可以被下载——请求的客户机通过网络从服务器上下载。它定义了一个方法 `isDenied()`, 这个方法指出服务器是拒绝还是同意了客户端的 `Request`。接口 `Response` 的实现可能提供了一些其他方法。

`AbstractResponse` 是 `Response` 的一个抽象类的实现, 它提供了一个默认的 `isDenied()` 方法体实现。`AbstractResponse` 的子类包括 `CreateUserResponse`、`CreateModelResponse`、`MoveResponse` 和 `ViewResponse`。有些子类, 例如 `UnknownRequestResponse` 和 `UnknownUserResponse`, 是用来指出在处理一个 `Request` 的同时, 服务器上发生了一个应用程序错误。

服务器不会立即为一个给定的 `Request` 返回 `Response` 的情况有以下几种。一种情况是, 某种给定类型的 `Request` 没有相应的 `Response`, 没有回复的必要。另一种情况是因为 `Request` 已经在队列中排队等待处理。

11.3.5 CreateUserConsumer

当客户机向服务器发送一个 `CreateUserRequest` 的时候, 服务器就会返回一个 `CreateUserResponse`, 并通过 `com.croftsoft.apps.chat.client` 包中的 `CreateUserConsumer` 在客户机上进行处理。`CreateUserConsumer` 提供了通用的回调方法 `consume()`, 就像在接口 `Consumer` 定义的一样, 因此它可以异步地接收和处理来自于服务器的消息。


```

public void consume ( Object o )
////////////////////////////////////
{
    CreateUserResponse createUserResponse = ( CreateUserResponse ) o;

    if ( createUserResponse.isDenied ( ) )
    {
        throw new RuntimeException ( "CreateUserRequest denied" );
    }
    else
    {
        requestQueue.replace ( createModelRequest );
    }
}

```

在接收到服务器上 User 对象成功创建提示的时候，CreateUserConsumer 将会通过发送 CreateModelRequest 进行响应。我曾经调用了行为有些像“反应堆”的那些客户端类，它们是通过服务器消息的响应实现客户机-服务器协议逻辑的，这些服务器消息是异步从服务器上下载和处理的。“反应堆”可以实现在客户机和服务器之间交换的一个有序序列，而不需要使用长期处在活动状态的线程，因为这样的线程可能会被联网调用锁死。

11.3.6 Event

只要更新了服务器端的游戏状态，就会产生 com.croftsoft.apps.chat.event 包中接口 Event 的一个实例。Event 扩展了 Serializable，因此它可以由玩家客户端通过从服务器上下载。它定义了两个方法：getEventIndex() 和 setEventIndex()，这样客户端就可以使用 eventIndex 序号检测来自于服务器的 Event 是否已经丢失。接口 Event 的实现还提供了其他一些方法。

AbstractEvent 是 Event 的一个抽象类实现，它为 eventIndex 属性方法提供了默认的实现。AbstractEvent 的子类包括 CreateModelEvent、MoveEvent、NullEvent、RemoveModelEvent 和 TalkEvent。

11.3.7 ChatClient

com.croftsoft.apps.chat.client 包中的类 ChatClient 提供了将客户端 Request 实例上传到游戏服务器的联网代码。它也下载服务器消息，例如 Event 和 Response 对象，并将它们发送给适当的客户端类进行处理。

ChatClient 被设置为可以在单玩家本地模式或多玩家联网模式中运行。在单玩家本地模式中，是没有联网功能的，它会创建一个本地游戏服务器。因为与游戏服务器之间的通信是被封装在 ChatClient 内的，客户机的其他代码并不需要知道游戏服务器是在本地还是在远程。

```
package com.croftsoft.apps.chat.client;
```

```
[...]
```

```
public final class ChatClient
    implements Lifecycle
```

```
////////////////////////////////////
```

```

////////////////////////////////////
{

[...]

private static final long POLLING_PERIOD_MIN
    = 0;

```

最小的轮询周期为 0。这意味着在收到前面请求的响应以后，ChatClient 会立即再与服务 器进行联系。服务器可能会延迟它对请求的响应，以便降低有效轮询频率。在 HTTP 轮询中， 客户机在请求之间延迟 30 秒左右的时间，而服务器会立即响应客户的请求。在 HTTP Pulling 中，客户机不会在请求之间进行延迟，但是服务器可能会延迟它的响应，延迟的时间大约是 30 秒——如果没有等待客户端的消息的话。

```

[...]

private final Authentication    authentication;

private final PullRequest      pullRequest;

private final Queue            eventQueue;

private final Queue            requestQueue;

private final ChatServer       chatServer;

private final Looper           loop;

private final HttpClient       httpClient;

private final Map              classToConsumerMap;

private final QueuePuller      queuePuller;

```

authentication 向服务器唯一地标识客户机。PullRequest 会被不断重复地发送到服务器。 eventQueue 存储从服务器上下载的事件。requestQueue 存储要上传到服务器的请求。如果游戏 是单玩家模式的，chatServer 就会提供游戏服务器的一份本地副本。在单玩家模式中，loop 将消息拖出本地游戏服务器。在多玩家模式中，httpClient 从远程游戏服务器上下载消 息。classToConsumerMap 将一个服务器类型映射到可以处理这些类型的客户端代码中。在单玩 家模式中，queuePuller 连续地将请求从 requestQueue 传送给 chatServer，并处理所有响应。

```
private long eventIndex;
```

最近处理事件的 eventIndex 是以实例变量的形式存储的。

```

public ChatClient (
    Authentication authentication,
    URL              servletURL )
{
    //////////////////////////////////

```



```

NullPointerException.check (
    this.authentication = authentication );
pullRequest = new PullRequest ( authentication );

if ( servletURL == null )
{
    requestQueue = new ListQueue ( );

    chatServer = new ChatServer ( );

    queuePuller = new QueuePuller (
        requestQueue,
        new Consumer ( )
        {
            public void consume ( Object o )
            {
                ChatClient.this.consume ( chatServer.serve ( o ) );
            }
        }
    );

    looper = new Looper (
        new Loopable ( )
        {
            public boolean loop ( )
            {
                return ChatClient.this.loop ( );
            }
        }
    );

    httpMessageClient = null;
}

```

当 `servletURL` 为空的时候，`chatClient` 就会运行在单玩家本地模式下。在客户机内存中会创建一个本地 `chatServer`，并且个别线程的循环也被确定下来，以便在客户机和服务器之间移动队列的消息。

```

else
{
    httpMessageClient = new HttpMessageClient (
        servletURL,
        USER_AGENT,
        SerializableCoder.INSTANCE,
        SerializableCoder.INSTANCE,
        HttpLib.APPLICATION_OCTET_STREAM, // contentType
        createRequestBytes ( pullRequest ),
        new Consumer ( )
        {
            public void consume ( Object o )
            {
                ChatClient.this.consume ( o );
            }
        }
    ),
}

```

```

        POLLING_PERIOD_MIN,
        POLLING_PERIOD_MAX,
        POLLING_PERIOD_INIT,
        POLLING_PERIOD_MULT,
        POLLING_PERIOD_DIVI,
        POLLING_PERIOD_INCR );

    requestQueue = httpMessageClient.getOutgoingQueue ( );

    chatServer = null;

    looper = null;

    queuePuller = null;
}

```

当 `servletURL` 不为空的时候，就会将 `httpMessageClient` 实例化，以便提供与远程游戏服务器之间的通信。Final 实例变量 `chatServer`、`looper` 和 `queuePuller` 被设置为空，因为在多玩家模式中没有使用它们。

```

    eventQueue = new ListQueue ( );

    Consumer eventConsumer = new QueueConsumer ( eventQueue );

    classToConsumerMap = new HashMap ( );

    classToConsumerMap.put (
        CreateModelResponse.class,
        NullConsumer.INSTANCE );

    classToConsumerMap.put (
        CreateUserResponse.class,
        new CreateUserConsumer (
            requestQueue,
            authentication,
            ChatConstants.DEFAULT_AVATAR_TYPE,
            ChatConstants.DEFAULT_AVATAR_X,
            ChatConstants.DEFAULT_AVATAR_Y ) );

    classToConsumerMap.put (
        CreateModelEvent.class,
        eventConsumer );

    classToConsumerMap.put (
        MoveEvent.class,
        eventConsumer );

    classToConsumerMap.put (
        NullEvent.class,
        NullConsumer.INSTANCE );

    classToConsumerMap.put (
        RemoveModelEvent.class,

```



```

eventConsumer );

classToConsumerMap.put (
    TalkEvent.class,
    eventConsumer );

classToConsumerMap.put (
    ViewResponse.class,
    eventConsumer );

```

ChatClient 从游戏服务器上下载消息。ChatClient 不是直接处理这些消息，而是将这些消息直接发送一个独立的类，以便于按照消息的不同类型进行相应的处理。通过分隔开不相关的逻辑，防止了 ChatClient 类随着时间的推移、游戏功能的增加变得太大而难于维护。

可以使用多重 if-then 语句或 switch 语句将消息发送给它们各自的目标以进行处理，但是 ChatClient 使用的是一个 HashMap。将消息对象的类作为键，而将消息处理器对象作为值。所有消息处理对象都实现了接口 Consumer，这样消息对象就可以使用通用的 consume() 方法处理。以后如果向 ChatClient 添加一个额外的 Consumer 实现，处理新类型的服务器消息，就会像在 classToConsumerMap 中添加一个键-值对一样简单。

来自于服务器但又被忽略的消息会被发送到一个 NullConsumer 的单态实例中。QueueConsumer 是一个适配器类，它在一个队列中存储已被消耗的对象。eventConsumer 是一个在 eventQueue 中存储来自于服务器的事件消息的 QueueConsumer。CreateUserResponse 类的消息是直接定向到 CreateUserConsumer 进行处理的。

```

eventIndex = new Random ( ).nextLong ( );
}

```

构造函数的最后任务是将 eventIndex 初始化为一个随机值。这就不太可能会将接收事件作为副本而丢弃。

```

public void init ( )
///////////////////////////////////////////////////
{
    if ( chatServer != null )
    {
        chatServer.init ( );

        looper.init ( );

        queuePuller.init ( );
    }
    else
    {
        httpMessageClient.init ( );
    }
}

```

如果客户机是在单玩家模式下操作，ChatClient 中的 init() 方法和其他 3 个生命周期方法都会委托给 chatServer、looper 和 queuePuller 中相应的方法。如果客户机是在多玩家联网模式下操作，生命周期方法就会委托给 httpMessageClient。

```
[...]

private boolean loop ( )
////////////////////////////////////
{
    Object response = chatServer.serve ( pullRequest );

    if ( response != null )
    {
        consume ( response );
    }

    return true;
}
```

在单玩家模式中，消息是直接从本地 chatServer 实例中拖出的。

```
private void consume ( Object o )
////////////////////////////////////
{
    [...]

    if ( o instanceof Event )
    {
        long eventIndex = ( ( Event ) o ).getEventIndex ( );

        if ( eventIndex == this.eventIndex )
        {
            return;
        }

        if ( eventIndex == this.eventIndex + 1 )
        {
            this.eventIndex = eventIndex;
        }
        else
        {
            requestQueue.replace ( new ViewRequest ( authentication ) );
        }
    }
}
```

consume()方法将每一个 Event 的 eventIndex 与前面已经处理过 Event 的 eventIndex(以实例变量的形式存储的)进行对比。如果 eventIndex 的值相同，那么新 Event 就会被作为一个副本而丢弃。如果新 eventIndex 就是新的而确实不是原来的 eventIndex，那么照常递增更新并存储新值。如果新 eventIndex 是其他的值，那么就可能是服务器发送到客户端的某些 Event 实例丢失了。在这种情况下，就会产生一个新的 ViewRequest。

```
else if ( o instanceof ViewResponse )
{
    this.eventIndex = ( ( ViewResponse ) o ).getEventIndex ( );
}
```


如果已经消耗的对象是 `ViewResponse` 的一个实例，那么就会更新 `eventIndex`，因为正在完全刷新模型的客户端副本。

```

Consumer consumer
    = ( Consumer ) classToConsumerMap.get ( o.getClass ( ) );

if ( consumer == null )
{
    [...]

    stop ( );
}
else
{
    try
    {
        consumer.consume ( o );
    }
    catch ( Exception ex )
    {
        [...]

        stop ( );
    }
}

[...]

```

来自于游戏服务器的事件和响应，无论是本地的还是远程的，都必须定向到适当的 `Consumer` 中进行处理。如果不能从 `classToConsumerMap` 上检索到 `Consumer`，或者出现什么问题，`ChatClient` 就会停止下来。当过时的客户软件从更新的游戏服务器上接收一种新类型的消息时，就会发生这种事情。在这种情况下，我认为停止客户端-服务器之间联网比打印一些错误消息和试图强制运行要好。

11.3.8 ChatSynchronizer

`ChatSynchronizer` 类存在于 `com.croftsoft.apps.chat.view` 包中，因为它是客户端视图的一部分。这个类基于从游戏服务器上接收到的事件，负责更新本地客户端游戏状态的显示。该类在玩家化身视图的区域限制以内，同步游戏状态的客户端副本和共享的服务器版本。

```

package com.croftsoft.apps.chat.view;

[...]

public final class ChatSynchronizer
    implements Consumer
{
    //////////////////////////////////////
    //////////////////////////////////////
    {

```

```
private final Consumer eventConsumer;
```

```
private final ChatWorld chatWorld;
```

ChatSynchronizer 维护 chatWorld 的一个单独的镜像副本。就像服务器上的 chatWorld 实例一样，在修改的时候，这个复制的 chatWorld 就产生状态更新事件。这些复制事件会传送给 eventConsumer。

```
[...]
```

```
public void consume ( Object o )
////////////////////////////////////
{
    if ( o instanceof ViewResponse )
    {
        ChatWorld newChatWorld
            = ( ( ViewResponse ) o ).getSeriChatWorld ( );

        chatWorld.clear ( );

        Model [ ] models = newChatWorld.getModels ( );

        for ( int i = 0; i < models.length; i++ )
        {
            ChatModel chatModel = ( ChatModel ) models [ i ];

            chatModel.setEventConsumer ( eventConsumer );

            chatWorld.addChatModel ( chatModel );
        }

        return;
    }
}
```

consume()方法接收并处理需要保持 chatWorld 的本地副本与游戏服务器实例同步的这些服务器消息。通常，这是通过使用在服务器产生的事件中发现的信息递增完成的。但是，偶尔也需要对整个 chatWorld 进行完全的刷新。当客户机第一次启动和发送一个初始 ViewRequest 的时候，或者在自从丢失了某些事件以后客户网络就一直处在错误状态时，就需要对整个 chatWorld 进行完全地刷新。在这些情况下，客户机将处理 ViewResponse 以更新整个 chatWorld，就像前面给出的代码一样。

```
if ( o instanceof MoveEvent )
{
    MoveEvent moveEvent = ( MoveEvent ) o;

    ChatModel chatModel
        = chatWorld.getChatModel ( moveEvent.getModelId ( ) );

    if ( chatModel == null )
    {
        return;
    }
}
```



```

    }

    PointXY origin = moveEvent.getOrigin ( );

    if ( origin != null )
    {
        chatModel.setCenter ( origin.getX ( ), origin.getY ( ) );
    }

    chatModel.setDestination ( moveEvent.getDestination ( ) );

    return;
}

if ( o instanceof CreateModelEvent )
{
    ChatModel chatModel
        = ( ( CreateModelEvent ) o ).getChatModel ( );

    chatModel.setEventConsumer ( eventConsumer );

    chatWorld.addChatModel ( chatModel );

    return;
}

if ( o instanceof RemoveModelEvent )
{
    ModelId modelId = ( ( RemoveModelEvent ) o ).getModelId ( );

    ChatModel chatModel = chatWorld.getChatModel ( modelId );

    chatModel.setActive ( false );

    return;
}
}

```

但是在大部分时间里，都可以使用来自于服务器产生的 MoveEvent、CreateModelEvent 和 RemoveModelEvent 的实例，递增地更新 chatWorld。注意，处理 CreateModelEvent 的时候，创建的 chatModel 新实例是和 eventConsumer 一起提供的，这样就可以观察对游戏状态本地视图的更改。

11.3.9 ChatPanel

ChatPanel 是主视图类。它提供了一个 Panel，这个 Panel 包含虚拟世界的一个动画视图和用于显示和输入文本消息的文本区域。

```

package com.croftsoft.apps.chat.view;

[...]

public final class ChatPanel

```

```

    extends JPanel
    implements ComponentAnimator, Lifecycle
    //////////////////////////////////////
    //////////////////////////////////////
{
    [...]

    private final Queue                eventQueue;

    private final ChatController        chatController;

    private final ChatGame chatGame;

    private final ChatSynchronizer      chatSynchronizer;

    [...]

    private final ChatGameAnimator      chatGameAnimator;

    private final AnimatedComponent      animatedComponent;

    [...]

```

`eventQueue` 保存从游戏服务器上下载的事件。维护指向 `chatController` 的引用，这样产生用户输入事件的各种不同的 `ChatPanel` 组件就可以在初始化过程中自动在 `chatController` 上进行注册。`chatGame` 是服务器端游戏状态的客户端镜像。`chatSynchronizer` 使用来自于 `eventQueue` 的事件更新 `chatGame`。`ChatPanel` 实现了 `ComponentAnimator`，这样它就可以通过委托给 `chatGameAnimator` 驱动 `animatedComponent`。上面并没有显示 `ChatPanel` 中的其他实例变量，包括 GUI 组件，例如文本区域。

```

public ChatPanel (
    Queue                eventQueue,
    ChatController chatController )
    //////////////////////////////////////
{
    super ( new BorderLayout ( ) );

    NullPointerException.check ( this.eventQueue = eventQueue );

    NullPointerException.check (
        this.chatController = chatController );

    chatGame = new SeriChatGame ( );

    chatSynchronizer = new ChatSynchronizer (
        chatGame.getChatWorld ( ),
        NullConsumer.INSTANCE );

    [...]

    animatedComponent = new AnimatedComponent ( this );

```



```
[...]

chatGameAnimator = new ChatGameAnimator (
    chatGame,
    animatedComponent,
    getClass ( ).getClassLoader ( ),
    ChatConstants.MEDIA_DIR );

[...]
}
```

构造函数用一个 `NullConsumer` 为它的 `eventConsumer` 参数初始化 `ChatSynchronizer` 实例。这意味着，由于对 `ChatWorld` 的本地副本的更新而产生的所有事件都将会被忽略。如果希望每次添加模型或将模型从游戏世界的客户端副本中删除的时候，都播放一个声音，那么就需要用一个不同的 `Consumer` 实现来替换这个 `NullConsumer`。

```
public void init ( )
///////////////////////////////////////////////////
{
    animatedComponent.init ( );

    animatedComponent.addComponentListener (
        new ComponentAdapter ( )
        {
            public void componentResized (
                ComponentEvent componentEvent )
            {
                animatedComponent.repaint ( );
            }
        } );

    chatController.setAvatarMenu ( avatarMenu );

    chatController.setMoveComponent ( animatedComponent );

    chatController.setMusicCheckBoxMenuItem ( musicCheckBoxMenuItem );

    chatController.setTalkTextField ( textField );
}

[...]
```

`init()` 方法初始化 `animatedComponent`，当面板的大小发生改变的时候，就设置一个自动完成重绘请求，并在 `chatController` 中注册那些产生用户输入事件的 GUI 组件。另外 3 个生命周期方法简单地委托给了 `animatedComponent`。

```
public void update ( JComponent component )
///////////////////////////////////////////////////
{
    chatGame.update ( );
}
```

```

Object o;

while ( ( o = eventQueue.poll ( ) ) != null )
{
    if ( o instanceof TalkEvent )
    {
        logPanel.record ( ( ( TalkEvent ) o ).getText ( ) );
    }
    else
    {
        chatSynchronizer.consume ( o );
    }
}

chatGameAnimator.update ( component );
}

[...]
```

服务器产生的事件都存储在 `eventQueue` 中，直到在游戏循环的更新阶段可以处理为止。在这种情况下，客户端游戏循环视图会镜像服务器游戏循环。例如，如果来自于服务器的一个 `MoveEvent` 指出某一个化身是在虚拟空间里一个给定的位置上，并正在向着一个具体的目标移动，`chatGame` 客户端副本的周期性更新就会递增地更新规划路线上的位置，而不需要来自于服务器的其他信息。

注意，`TalkEvent` 实例是由 `ChatPanel` 进行处理的，但是所有其他事件都将转发到 `chatSynchronizer`，因为这些事件包含对虚拟游戏世界的变更。

11.3.10 ChatApplet

`com.croftsoft.apps.chat`包中的`ChatApplet`是包含模型、视图和控制器的主要客户端类。

```

package com.croftsoft.apps.chat;

[...]

public final class ChatApplet
    extends JApplet
    //////////////////////////////////////
    //////////////////////////////////////
{
    [...]

    private ChatClient chatClient;

    private ChatPanel chatPanel;
```

维护指向 `chatClient` 和 `chatPanel` 的引用，这样就可以使用生命周期方法将联网和动画线程挂起和恢复。

[...]

```
public void init ( )
///////////////////////////////////////////////////
{
    System.out.println ( getAppletInfo ( ) );

    // model

    Authentication authentication = getAuthentication ( );

    chatClient = new ChatClient ( authentication, getServletURL ( ) );
```

chatClient 将模型表示出来。如果私有模型 getServletURL() 返回 null, 那么客户端就会运行在单玩家本地模式中, 并会在本地内存中创建一个游戏服务器对象。否则, chatClient 将会使用 HTTP Pulling 与远程游戏服务器建立一个网络连接。注意, 玩家的用户名称和密码是使用私有方法 getAuthentication() 检索的。

```
// controller

ChatController chatController = new ChatController (
    authentication, chatClient.getRequestQueue ( ) );
```

chatController 使用玩家的 authentication 创建 Request 对象, 其中, 玩家可以被服务器唯一地标识和认证。请求在由 chatClient 提供的 requestQueue 队列中排队, 以便于稍后上传到游戏服务器。

```
// view

chatPanel = new ChatPanel (
    chatClient.getEventQueue ( ) , chatController );

setJMenuBar ( chatPanel.getMenuBar ( ) );

Container contentPane = getContentPane ( );

contentPane.setLayout ( new BorderLayout ( ) );

contentPane.add ( chatPanel, BorderLayout.CENTER );

LifecycleLib.init ( chatPanel );

validate ( );

LifecycleLib.init ( chatClient );
}
```

该代码创建了用户接口视图。ChatPanel 构造函数的参数包括由 chatClient 创建的 eventQueue(这个参数用做输入)和 chatController(这个参数用做输出)。ChatPanel 视图类使用

chatController 来间接地将消息发送到游戏服务器——只要发生用户输入事件。ChatPanel 使用 chatClient 模型类中的 eventQueue 来观察和反射服务器端的游戏状态更新事件。

这个视图是在包含联网代码的 chatClient 之前进行初始化的。其他 3 个生命周期方法只是简单地委托给 chatClient。

```
private static Authentication getAuthentication ( )
///////////////////////////////////////////////////
{
    String username = Long.toString ( new Random ( ).nextLong ( ) );

    return new Authentication ( username, "password" );
}

[...]
```

私有的静态方法 getAuthentication()只是简单地产生一个随机的用户名，以便向服务器惟一标识玩家客户端实例。当服务器上的 User 对象不具有持久性的时候，这非常有效。由于安全需求较低，所以我对这里的密码提供了一个哑元值。如果服务器上的 User 对象具有持久性，就需要更改 ChatPanel，以便提供用户名称和密码的文本输入字段。

11.4 服务器端代码

在游戏服务器上，User 对象存储某个特定玩家客户端的数据和消息。UserStore 提供存储和检索各个 User 对象的能力。UserStore 服务于从游戏服务器拖出消息的客户端请求。MoveServer 对在虚拟的游戏世界中移动化身的请求提供服务。SeriChatGame 包含游戏状态和逻辑。ChatServer 服务于客户端请求并驱动主游戏循环。

11.4.1 User

实现了包 com.croftsoft.apps.chat.User 中的接口 User 的类，存储某个具体玩家客户端的数据和消息。

```
package com.croftsoft.apps.chat.user;

import com.croftsoft.core.util.queue.Queue;

import com.croftsoft.core.animation.model.ModelId;

public interface User
///////////////////////////////////////////////////
///////////////////////////////////////////////////
{
    public long      getEventIndex      ( );

    public long      getLastRequestTime ( );

    public Queue     getMessageQueue    ( );
```



```

public ModelId getModelId      ( );

public String getPassword      ( );

public Queue  getRequestQueue  ( );

public UserId getUserId        ( );

public String getUsername      ( );

```

`eventIndex` 对发送到某一个具体 `User` 的事件进行连续编号。`lastRequestTime` 是上一次用户和服务器联系的时间。当游戏服务器为一个具体的 `User` 产生事件和响应对象的时候，这些事件和对象都会存储在 `messageQueue` 中，以便由玩家客户端使用 HTTP Pulling 进行检索。`User` 的 `modelId` 提供一个指向虚拟世界里相应化身的持久性引用。当玩家客户端向服务器发送一个请求的时候，这个请求就会被存储在 `requestQueue` 中，以便于稍后进行处理。`User` 是使用 `UserId` 或 `username` 进行惟一标识的。

```

public long  nextEventIndex ( );

public void  setModelId ( ModelId modelId );

public void  setPassword ( String password );

public void  updateLastRequestTime ( );

```

`User` 的可变属性是 `eventIndex`、`modelId`、`password` 和 `lastRequestTime`。

11.4.2 UserStore

接口 `UserStore` 提供了一种存储和检索 `User` 对象的方法。它可以使用持久存储器来实现这个接口，也可以不用持久存储器实现这个接口。

```

package com.croftsoft.apps.chat.user;

import com.croftsoft.core.security.Authentication;

public interface UserStore
////////////////////////////////////
////////////////////////////////////
{

    public User      getUser      ( Authentication authentication );

    public User      getUser      ( UserId userId );

    public UserId    getUserId    ( String username );

    public UserId [ ] getUserIds  ( );
}

```

```

////////////////////////////////////
////////////////////////////////////

```

```
public UserId    createUser ( Authentication authentication );
```

```
public boolean  removeUser ( UserId userId );
```

接口 `UserId`、`User` 和 `UserStore` 的实现包括 `com.croftsoft.apps.chat.user.seri` 包中的 `SeriUserId`、`SeriUser` 和 `SeriUserStore`。

11.4.3 PullServer

`com.croftsoft.apps.chat.server` 包中的 `PullServer` 对客户端从服务器上拖出消息的请求提供服务。

```
package com.croftsoft.apps.chat.server;
```

```
[...]
```

```
public final class PullServer
    extends AbstractServer
```

```

////////////////////////////////////
////////////////////////////////////
{

```

```

    public static final long DEFAULT_QUEUE_PULL_TIMEOUT
        = 30 * MathConstants.MILLISECONDS_PER_SECOND;

```

拖出一个消息默认的超时值是 30 秒。这可以变得更长一些，以便当虚拟游戏世界中没有发生什么事情的时候，减少客户机和服务器之间的网络流量。但是如果太长的话，客户机可能不会注意到由于网络的失败而使它和服务器在某个时候断开了连接。超过 30 秒后，客户端可以得到服务器的某种“保证”：连接仍然存在——只是没有产生什么事件。

```

public Object serve (
    User      user,
    Request   request )
{
    Queue messageQueue = user.getMessageQueue ( );

    Object message = null;

    try
    {
        message = messageQueue.pull ( queuePullTimeout );
    }
    catch ( InterruptedException ex )
    {
    }

    if ( message == null )
    {
        message = new NullEvent ( user.getEventIndex ( ) );
    }
}

```



```

    }
    else if ( message instanceof Event )
    {
        ( ( Event ) message ).setEventIndex ( user.nextEventIndex ( ) );
    }

    return message;
}

```

PullServer 将会试图从 messageQueue 中为某个具体的 User 拖出消息。如果某个消息在到达超时期限以前变为可用，或者这个消息已经在 messageQueue 中等待，serve() 就会立即用一个递增的 eventIndex 返回这个消息。否则，它就会在队列上进行拖拉直到到达超时期限，然后再用最新的 eventIndex 返回一个 NullEvent。当发出 NullEvent 的时候，eventIndex 并不会递增，因为客户端的游戏状态同步算法并不关心在传输过程中 NullEvent 是否丢失。

11.4.4 MoveServer

MoveServer 对在虚拟世界里移动化身的客户端请求提供服务。它也作为修改服务器端游戏状态的服务器端代码的一个示例。

```

package com.croftsoft.apps.chat.server;

[...]

public final class MoveServer
    extends AbstractServer
    //////////////////////////////////////
    //////////////////////////////////////
    {

    private final ChatWorld chatWorld;

    //////////////////////////////////////
    //////////////////////////////////////

    public MoveServer ( ChatWorld chatWorld )
    //////////////////////////////////////
    {
        NullArgumentException.check ( this.chatWorld = chatWorld );
    }
}

```

MoveServer 操纵 ChatWorld。

```

public Object serve (
    User    user,
    Request request )
    //////////////////////////////////////
    {
        MoveRequest moveRequest = ( MoveRequest ) request;

        ModelId modelId = user.getModelId ( );
    }
}

```

```

    if ( modelId == null )
    {
        return new MoveResponse ( true, true, false );
    }
    ChatModel chatModel = chatWorld.getChatModel ( modelId );

    if ( chatModel == null )
    {
        return new MoveResponse ( true, false, true );
    }

```

如果有某个请求要修改一个还不存在的化身，就会返回一个应用程序错误的响应。**MoveResponse** 的构造函数接收表示拒绝了这个请求的布尔标志，因为 **ModelId** 未知，或者是不能检索到相应的 **ChatModel** 实例。

```

    chatModel.setDestination ( moveRequest.getDestination ( ) );

    return null;
}

```

chatModel 的目标被更新，但是没有返回 **MoveResponse**。在这里，只有请求被拒绝的时候才会返回 **MoveResponse**。注意，当 **chatModel** 的目标被更新的时候，**chatModel** 就会向带有视图重叠区域的所有客户端广播状态更新事件。

11.4.5 SeriChatGame

com.croftsoft.apps.chat.model.seri 包中的 **SeriChatGame** 类包含游戏的状态和逻辑。

```

package com.croftsoft.apps.chat.model.seri;

[... ]

public final class SeriChatGame
    implements Serializable, ChatGame
    //////////////////////////////////////
    //////////////////////////////////////
    {

```

SeriChatGame 实现了 **com.croftsoft.apps.chat.model** 包中的接口 **ChatGame**。它还实现了 **Serializable**，这样当应用程序服务器关闭的时候，游戏的状态就可以保存到磁盘。但是在这个示例中并没有实现这种持久性机制。

```

[... ]

private final UserStore  userStore;

private final ChatWorld  chatWorld;

private final Timekeeper timekeeper;

private final Map        classToRequestServerMap;

```


`classToRequestServerMap` 将 `Request` 对象映射给 `Server` 对象，而后者可以基于 `Request` 的类处理它们。

[...]

```
public SeriChatGame ( UserStore userStore )
////////////////////////////////////
{
    NullPointerException.check ( this.userStore = userStore );

    Consumer eventConsumer =
        new Consumer ( )
        {
            public void consume ( Object o )
            {
                broadcast ( o );
            }
        };

    chatWorld = new SeriChatWorld ( eventConsumer );
```

`chatWorld` 模型状态的改变会产生状态更新事件。这些事件被传递到 `eventConsumer`，而 `eventConsumer` 再将它们委托给私有的 `broadcast()` 方法。

```
timekeeper = new Timekeeper ( SystemClock.INSTANCE, TIME_FACTOR );
```

`timekeeper` 是用来确定模拟更新之间的时间增量的。

```
classToRequestServerMap = new HashMap ( );

classToRequestServerMap.put (
    CreateModelRequest.class,
    new CreateModelServer ( chatWorld ) );

classToRequestServerMap.put (
    MoveRequest.class,
    new MoveServer ( chatWorld ) );

classToRequestServerMap.put (
    TalkRequest.class,
    new TalkServer ( eventConsumer ) );

classToRequestServerMap.put (
    ViewRequest.class,
    new ViewServer ( ( SeriChatWorld ) chatWorld ) );
}
```

`SeriChatGame` 再在游戏循环的更新阶段，通过将 `User` 请求委托给 `CreateModelServer`、`MoveServer`、`TalkServer` 或 `ViewServer` 的方法，处理这些请求。

[...]

```
public void update ( )
```

```

////////////////////////////////////
{
    chatWorld.prepare ( );

    Model [ ] models = chatWorld.getModels ( );

    for ( int i = 0; i < models.length; i++ )
    {
        Model model = models [ i ];

        if ( !model.isActive ( ) )
        {
            chatWorld.removeModel ( model.getModelId ( ) );
        }
    }
}

```

非活动的模型从 chatWorld 中删除。当从服务器上的 SeriChatGame 中删除模型的时候，客户机上 SeriChatGame 中相应的副本在第一次更新时就被标志为非活动的，而在第二次更新时就会通过上面的代码将其删除。这就为客户端视图的动画代码提供一个机会，利用这个机会可以观察从活动状态到非活动状态的转变，并在模型和它的驱动者被永久销毁以前，使用已经删除的化身重绘原来精灵区域。如果没有这个特性，化身的原图像就可能一直残留在屏幕上，即使已从虚拟世界中将它的模型删除了。

```

UserId [ ] userIds = userStore.getUserIds ( );

for ( int i = 0; i < userIds.length; i++ )
{
    User user = userStore.getUser ( userIds [ i ] );

    if ( user == null )
    {
        continue;
    }

    if ( System.currentTimeMillis ( )
        >= user.getLastRequestTime ( ) + ChatConstants.USER_TIMEOUT )
    {
        removeUser ( user );

        continue;
    }
}

```

而最近还没有和游戏服务器进行联系的玩家客户端的 User 对象就会从游戏中删除。

```

Queue requestQueue = user.getRequestQueue ( );

Request request = ( Request ) requestQueue.poll ( );

```



```

if ( request == null )
{
    continue;
}

RequestServer requestServer = ( RequestServer )
    classToRequestServerMap.get ( request.getClass ( ) );

if ( requestServer == null )
{
    queue ( user, new UnknownRequestResponse ( request ) );
}

Object response = requestServer.serve ( user, request );
if ( response != null )
{
    queue ( user, response );
}
}

```

在更新过程中，会对每个 User 的一个请求进行服务。如果有响应，就会针对具体 User 使用私有的 queue()方法，将响应在 messageQueue 中排队。

```

timekeeper.update ( );

double timeDelta = timekeeper.getTimeDelta ( );

if ( timeDelta > TIME_DELTA_MAX )
{
    timeDelta = TIME_DELTA_MAX;
}

chatWorld.update ( timeDelta );
}

```

先将 chatWorld 准备好，再对 User 请求进行服务，最后更新 chatWorld。

```

private void queue (
    User    user,
    Object  message )
////////////////////////////////////
{
    Queue messageQueue = user.getMessageQueue ( );

    try
    {
        messageQueue.replace ( message );
    }
}

```

```

        catch ( IndexOutOfBoundsException ex )
        {
            removeUser ( user );
        }
    }
}

```

私有方法 `queue()` 使用合并的 `replace()` 方法，为 `User` 将消息插入 `messageQueue`。如果 `messageQueue` 已满，它就会抛出 `IndexOutOfBoundsException` 异常。在这种情况下，就认为队列已经被填满了，可能因为玩家客户机不再从服务器上拖消息了，也可能是因为客户机从服务器上下载消息的速度不够快，以至于跟不上填充消息的速度。这个问题通过从游戏中删除 `User` 对象的方法可以得到解决。如果玩家客户机与服务器重新建立了连接，它就需要发送一个 `ViewRequest` 以获得一个完全状态的刷新，因为增加的更新事件已经丢失了。

```

private void broadcast ( Object o )
///////////////////////////////////////////////////
{
    UserId [ ] userIds = userStore.getUserIds ( );

    for ( int i = 0; i < userIds.length; i++ )
    {
        User user = userStore.getUser ( userIds [ i ] );

        if ( user == null )
        {
            continue;
        }

        queue ( user, o );
    }
}

```

在 `chatWorld` 中发生一个游戏状态事件时，这个游戏状态事件会被发送到所有带有视图重叠区域的玩家客户机上。在这个游戏示例中，为了简单起见，每个玩家都可以看到所有的事件。

```

private void removeUser ( User user )
///////////////////////////////////////////////////
{
    userStore.removeUser ( user.getUserId ( ) );

    ModelId modelId = user.getModelId ( );

    if ( modelId != null )
    {
        chatWorld.removeModel ( modelId );
    }
}

```

从游戏中删除一个 `User` 并不产生状态更新事件，但是从 `chatWorld` 中删除它相应的化身 `Model` 则会产生状态更新事件。

11.4.6 ChatServer

ChatServer 服务于客户端请求并驱动游戏主循环以更新服务器端的游戏状态。在多玩家联网模式中，ChatServer 驻留在一个远离玩家客户机的游戏服务器上。它是通过 ChatServlet 间接进行访问的，该 ChatServlet 转换并传播来自于玩家客户机的 HTTP 请求。在单玩家模式中，ChatServer 以 ChatClient 的一个实例变量的形式驻留在本地内存中，使用一个方法调用就可以直接访问它。

```
package com.croftsoft.apps.chat.server;

[...]

public final class ChatServer
    implements Commissionable, Server
    //////////////////////////////////////
    //////////////////////////////////////
    {

    [...]

    public ChatServer ( )
    //////////////////////////////////////
    {
        userStore = new SeriUserStore ( );

        chatGame = new SeriChatGame ( userStore );

        createUserServer = new CreateUserServer ( userStore );

        pullServer = new PullServer ( ChatConstants.QUEUE_PULL_TIMEOUT );

        looper = new Looper (
            new Loopable ( )
            {
                public boolean loop ( )
                {
                    return ChatServer.this.loop ( );
                }
            },
            new FixedDelayLoopGovernor ( UPDATE_RATE ),
            null,
            ( String ) null,
            Thread.MIN_PRIORITY,
            true );
    }
}
```

构造函数创建一个在内存中的新 userStore 实例，这个 userStore 实例最开始是空的。然后它创建了一个 chatGame 实例，该实例作为服务器端的游戏状态使用。CreateUserServer 对创建一个 User 对象的请求提供服务。looper 驱动主循环线程，这个主循环线程周期性地更新游戏状态以在虚拟世界中保持物体的移动。

```

public void init ( )
///////////////////////////////////////////////////
{
    startTime = System.currentTimeMillis ( );

    lastRequestTime = startTime;

    looper.init ( );
}

public void destroy ( )
///////////////////////////////////////////////////
{
    looper.stop ( );

    looper.destroy ( );
}

```

方法init()和destroy()委托给looper。这里没有start()和stop()这两个生命周期方法。

```

public Object serve ( Object requestObject )
///////////////////////////////////////////////////
{
    [...]

    looper.start ( );

    if ( requestObject instanceof CreateUserRequest )
    {
        CreateUserRequest createUserRequest
            = ( CreateUserRequest ) requestObject;

        return createUserServer.serve ( createUserRequest );
    }
}

```

创建一个用户的请求必须同步地得到服务。一旦创建了一个 User 对象，其他类型的请求就可以存储在 requestQueue 中，等待该 User 稍后进行异步地处理。如果为这些其他类型的请求产生了一个响应，那么这个响应就可以为该 User 而存储 messageQueue 中。但是，对 CreateUserRequest 而言，响应必须同步地返回，因为请求可能会被拒绝。在这种情况下，将不会创建 User 对象，因此也不会创建可能已被用于 CreateUserResponse 失败消息的存储和异步传输的 messageQueue。

```

[...]

User user = userStore.getUser ( authentication );

if ( user == null )
{
    if ( request instanceof PullRequest )
    {
        return createUserServer.serve (

```

```

        new CreateUserRequest ( authentication ) );
    }

    return new UnknownUserResponse ( request );
}

```

如果在服务器上创建 User 对象以前，玩家客户机就发送了一个 PullRequest，那么这个 PullRequest 就会被当作是 CreateUserRequest。这就不需要客户机保证在它发出第一个 PullRequest 以前首先要发出一个 CreateUserRequest。

```

    user.updateLastRequestTime ( );

    if ( request instanceof PullRequest )
    {
        return pullServer.serve ( user, request );
    }

    user.getRequestQueue ( ).replace ( request );

    return null;
}

```

像 CreateUserRequest 一样，PullRequest 也是同步得到服务的。与 CreateUserRequest 不同，服务器可能不会立即响应 PullRequest，因为响应可能会被延迟——如果没有消息在 messageQueue 中等待 User 的话。其他类型的请求都会在 requestQueue 中进行排队，以便在稍后的游戏循环的更新阶段中，用带有使用 HTTP Pulling 机制异步返回的所有响应进行处理。

```

private boolean loop ( )
///////////////////////////////////////////////////
{
    chatGame.update ( );

    [...]

    return true;
}

```

游戏主循环更新服务器端的游戏状态。chatGame 实例的 update() 方法包含了为每一个 User 处理存储在 requestQueue 实例中的玩家客户机请求的逻辑。

11.5 跟踪消息

在这一章中，我想强调的最主要的一点就是，尽管 HTTP 一般都是专门用于单向同步请求，但还是可以使用这里介绍的 HTTP Pulling 技术，用它来模拟服务器和客户端之间事件驱动机制的异步通信。我想强调的第二点是，使用带有队列的异步通信机制后，客户机的代码是在本地还是在远程都没有关系。我认为很值得做的一个练习可能就是，无论客户端是在多玩家网络模式中，还是在单玩家本地模式中，都跟踪从客户端发送到服务器以及从服务器向客户端返回的消息。

11.5.1 多玩家联网模式

正如图 11-2 所示的一样，当用户单击正好处在视图(V)中的鼠标时，就会产生一个 MouseEvent(MSE)，并会将其存储在 AWT EventQueue(AEQ)中。稍后事件分派线程会从 EventQueue 中删除 MouseEvent，并通过 MouseListener 的接口方法 mousePressed()将它传递给 ChatController(CCO)。ChatController 将 MouseEvent 转换为 MoveRequest(MVR)并将它存储在发送队列 requestQueue (ORQ)中。

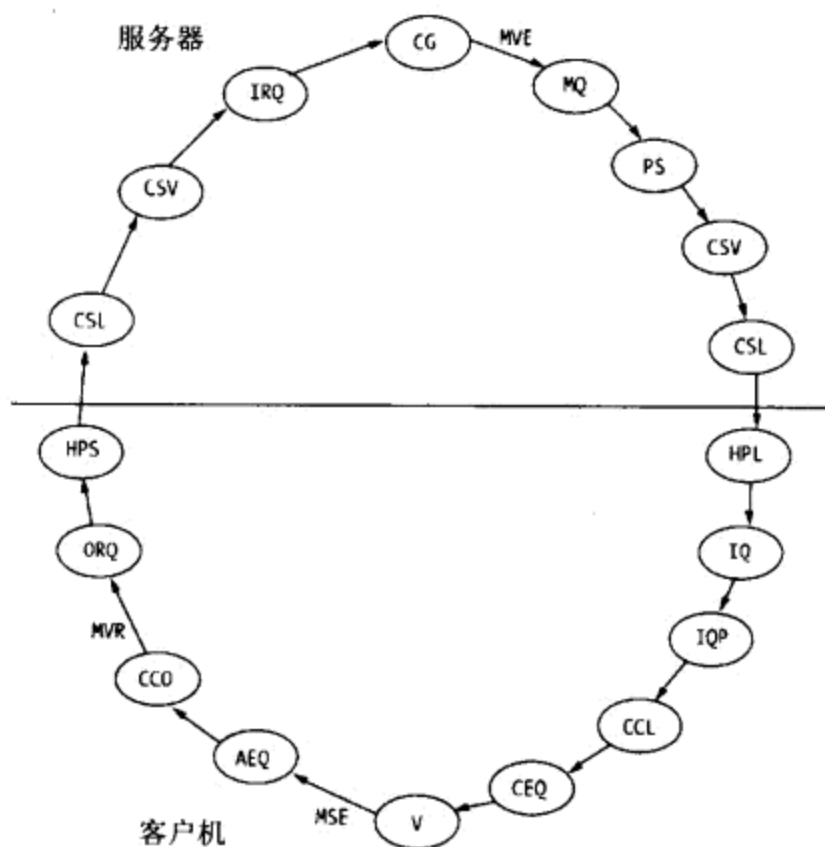


图 11-2 多玩家联网模式中的消息传递路径

外发的 requestQueue 对 HttpMessagePusher(HPS)而言是 outgoingQueue。HttpMessagePusher 使用 HTTP，将 outgoingQueue 里的对象连续地上传到 ChatServlet(CSL)。ChatServlet 转换以串行化并压缩的字节数组的形式发送出的、又返回到一个 MoveRequest 对象中的 HTTP 请求，并将它发送给 ChatServer(CSV)。ChatServer 在与产生这个请求的 User 关联的一个 requestQueue (IRQ)接收队列中存储这个 MoveRequest。在服务器端游戏循环的更新阶段中，来自于每一个 User 的 requestQueue 接收队列中的每一个请求就会被检索出来，并由 ChatGame (CG)进行处理。

当移动化身的时候，处理 MoveRequest 会导致服务器端游戏状态被更新。这产生了一个 MoveEvent(MVE)，该 MVE 是为游戏中所有 User，逐一存储在 messageQueue(MQ)发送队列中的。在将 MoveEvent 插入到 messageQueue 队列中时，该队列就会自动将当前可以检索出一个对象的这种情况，通知给在 pull()方法中可能已经被阻塞的所有线程。一旦将 MoveEvent 添加到队列中，阻塞在 messageQueue 队列里 pull()方法中的 PullServer (PS)线程就会被自动唤醒，并移除该 MoveEvent。当对象从 serve()方法中返回时，PullServer(PS)线程将 MoveEvent 传递给 ChatServer (CSV)。然后 ChatServer 再将它传递给调用它自己的 serve()方法的对象 ChatServlet(CSL)。ChatServlet 串行化并压缩这个 MoveEvent，再将它作为对来自于玩家客户端的 HTTP 请求延迟的响应返回。

HttpMessagePoller(HPL)下载、解压缩和反串行化 MoveEvent，并将它存储在 incomingQueue (IQ)中。HttpMessageClient 中的 incomingQueuePuller(IQP)立即从 incomingQueue 中移除这个

MoveEvent, 并通过 consume() 方法将它传递给 ChatClient(CCL)。ChatClient 将 MoveEvent 路由到 eventConsumer, 后者将 MoveEvent 插入到聊天客户端 eventQueue(CEQ)中。

在客户端视图的更新阶段中, ChatPanel(V)从 eventQueue 中移除 MoveEvent, 并将它传递给 ChatSynchronizer。ChatSynchronizer 使用 MoveEvent 中的信息更新它对游戏状态的备份。客户端的游戏状态, 也即模型镜像, 在更新的时候也会产生一个事件, 但是这个事件会被丢弃。

注意, 在环状消息路径上共有 6 个队列: AWT EventQueue(AEQ)队列、客户端 requestQueue(ORQ)发送队列、服务器端 requestQueue(IRQ)接收队列、服务器端 messageQueue(MQ)发送队列、客户端 incomingQueue (IQ)队列和聊天客户端 eventQueue(CEQ)队列。这些队列中的每一个队列都会在线程之间传递消息。如果没有这些队列, 任何网络或消息处理的延迟都会使整个系统的性能下降。另外, 在更新阶段通过消息串行化的处理避免了并发问题, 如果没有队列的话, 做到这一点将是非常困难的。

11.5.2 单玩家本地模式

正如图 11-3 所示的一样, 单玩家本地模式的消息流几乎和多玩家联网模式的消息流完全一样。发送队列 queuePuller(OQP)将请求直接传递给本地内存中 ChatServer(CSV)对象的 serve() 方法, 而不使用中介性质的 HttpMessagePusher (HPS)和 ChatServlet(CSL)。PullRequest 对象被直接传递给 ChatServer 而不是使用 HttpMessagePuller (HPL)或 ChatServlet (CSL)。来自于 ChatServer 的响应被直接发送给 ChatClient(CCL)的 consume() 方法, 而不是首先在 incomingQueue (IQ)中存储, 然后再使用 incomingQueuePuller(IQP)将其删除。

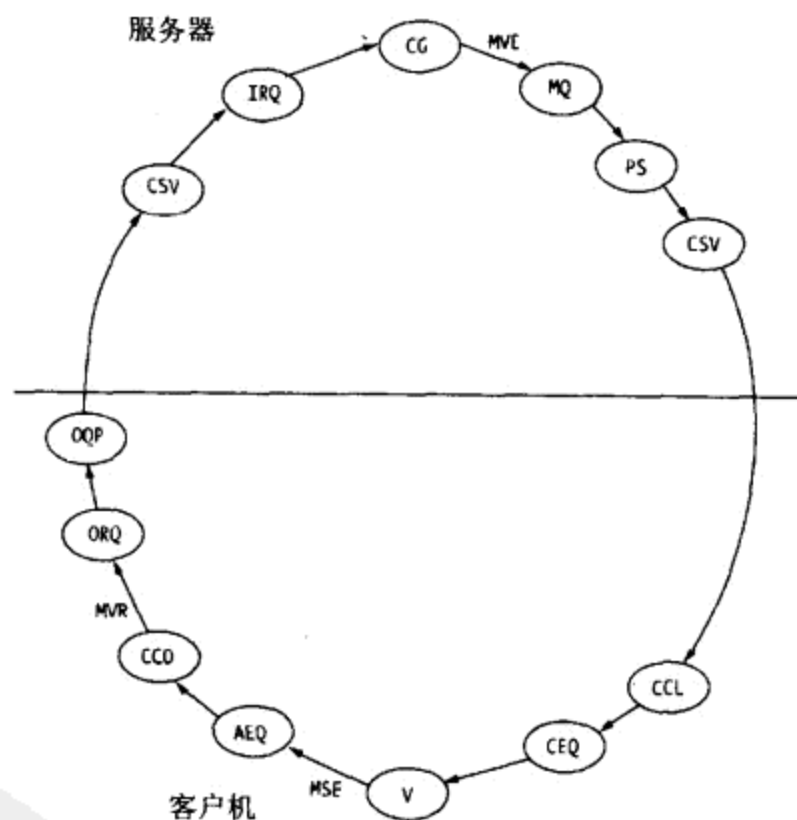


图 11-3 单玩家本地模式中的消息传递路径

使用队列和异步调用方法能够带来很多灵活性。因为 ChatController 是在发送队列 requestQueue 中存储请求消息, 而不是直接将消息传递给另一个对象进行上传或处理, 所以将 QueuePuller (OQP)和 HttpMessagePusher (HPS)进行交换是很容易的事情——因为它们都可以从队列中读取消息。在这个示例中, requestQueue 维护控制器和模型之间的松耦合, 不管模型是在远程还是在本地。同样, 由于 ChatClient 使用异步调用方法 consume()接收来自于模型的

消息，所以它不需要知道这些消息是直接来自于内存中的 ChatServer，还是通过一个网络层到达的。

11.6 扩展示例

通过创建、链接和映射适当的 Request、Server、Response、Event 和 Consumer 消息以及消息处理类，可以逐渐地将一些辅助功能添加到这个游戏中。例如，为了让化身可以捡起虚拟世界里的某一个东西，可能需要创建一个 PickUpItemRequest 类，只要玩家在视图里的一个条目上右键单击鼠标，ChatController 就会发送该请求。在服务器上，PickUpItemServer 将会处理这个请求，并返回一个 PickUpItemResponse。“捡起”这个条目肯定会产生一个服务器端游戏状态改变事件，这个事件应该发送给具有视图重叠区域的所有玩家客户端。这些客户端接着再使用 PickUpItemConsumer 异步处理 PickUpItemEvent。

这个实例会在简单的 servlet 容器(例如 Tomcat)内运行。但是我想，为了将这个示例游戏扩展为健壮的大型多玩家在线角色扮演游戏(massively multiplayer online role playing game, MMORPG)，也许需要使用支持 J2EE EJB(例如 JBoss)的应用程序服务器。这里介绍的消息传递和排队机制也可能需要由消息驱动的 bean(message-driven bean, MDB)、持久的订阅、主题和服务端的消息过滤和基于虚拟地理位置和视图区域的选择机制所替换。不管是否使用了 MDB，HTTP Pulling 网络层对于客户机-服务器之间在安全限制以内连续进行的通信都是必要的。对于游戏状态和用户数据持久性来说 CMP 优于简单地串行化到磁盘。一旦用户数据变为持久，联网安全性就需要更新。

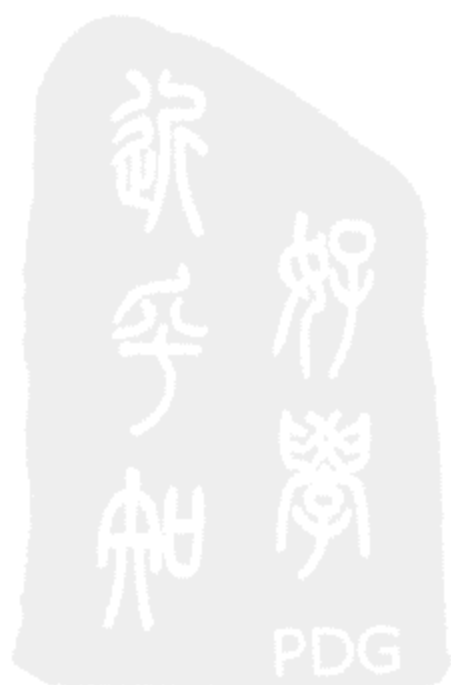
您可能会考虑为在游戏服务器和非 Java 客户机之间的基于 XML 的消息使用 JAXB。如果真的如此，可能还希望检查 JAXM 的适用性。业务处理企业综合可视化编程工具的主要用户就是具有很少或干脆就没有编程经验的业务分析师，因此任何人都可以指望它们能对游戏设计有所帮助。由于 J2EE 框架完全能够满足重大型企业处理的需要，所以我不会对它能很好地应用在 MMORPGs 上而感到大惊小怪。

11.7 小结

本章介绍了一个使用事件驱动通信机制而不是使用轮询机制的多玩家联网游戏示例。服务器发起的事件通知消息是使用 HTTP Pulling 技术进行模拟的，在这种技术中，服务器延迟它对 HTTP 客户端请求的响应，直到消息可用为止。客户机不需要任何其他通信就可以通过更新它基于以前下载的原始服务器事件的镜像模型，跟踪服务器端的游戏状态。因为引用不能在游戏服务器和客户机之间分布，所以使用 Id 实例来惟一标识分布式的对象，例如化身模型。只要切实可行，消息就会在队列中进行合并，以便减少网络通信量和所需要的处理。HashMap 实例用来将消息映射给可以处理它们的对象，而不是使用多重 if-then 语句进行处理。消息队列和异步通信的目的是提高灵活性，并防止由于联网和处理延迟导致的停止。通过递增地添加新消息传递和消息处理类，可以很容易地扩展这个示例游戏，但是开发一个 MMORPG 也许需要对集成的 EJB 进行一下检查。

11.8 参考文献

Chappell, David A. and Tyler Jewell. "J2EE and Web Services." Chapter 8 in *Java Web Services: Using Java in Service-Oriented Architectures*. Sebastopol, CA: O'Reilly & Associates, 2002.



附录 A 源代码索引

本书介绍了 150 多个新类。有些介绍得比较详细，而有些只是点到为止。其中的很多类都具有高可重用性。本附录给出了这些新类的一个列表，这个列表是按照包的名称和类的名称进行排列的。列表中列出的是静态方法或包而不是类。

第 1 列是在书中介绍这个类的章编号。如果知道类名称但是不知道包名称，那么可以使用在线 javadoc 来获取相应的包名称。如果某一个类在多个章中都介绍过，那么在此就可能会多次将它列出。

第 2 列是在本书中归档代码的日期，这个日期是通过源代码文件中 javadoc 标记@version 标出的。如果您所使用的是源代码库里的最新代码，而不是这个快照归档代码，那可能就是带有一些变更的新版本，而在本书中并没有介绍这些变更的新版本。

```
03 2003-05-06 com.croftsoft.ajgp.anim.ExampleAnimator
01 2003-11-06 com.croftsoft.ajgp.basics.BasicsExample
06 2003-03-12 com.croftsoft.ajgp.data.GameData
06 2003-03-13 com.croftsoft.ajgp.data.SerializableGameData
05 2003-08-01 com.croftsoft.ajgp.grap.FullScreenDemo
09 2003-06-04 com.croftsoft.ajgp.http.ScoreApplet
09 2003-06-04 com.croftsoft.ajgp.http.ScoreServlet
11 2003-06-17 com.croftsoft.apps.chat.ChatApplet
11 2003-08-16 com.croftsoft.apps.chat.client.ChatClient
11 2003-06-20 com.croftsoft.apps.chat.client.CreateUserConsumer
11 2003-06-17 com.croftsoft.apps.chat.controller.ChatController
11 2003-06-18 com.croftsoft.apps.chat.event.Event
11 2003-09-10 com.croftsoft.apps.chat.model.seri.SeriChatGame
11 2003-06-20 com.croftsoft.apps.chat.request.CoalesceableRequest
11 2003-06-10 com.croftsoft.apps.chat.request.Request
11 2003-06-17 com.croftsoft.apps.chat.response.Response
11 2003-06-18 com.croftsoft.apps.chat.server.ChatServer
11 2003-06-10 com.croftsoft.apps.chat.server.MoveServer
11 2003-06-18 com.croftsoft.apps.chat.server.PullServer
11 2003-06-18 com.croftsoft.apps.chat.user.User
11 2003-06-07 com.croftsoft.apps.chat.user.UserStore
11 2003-06-25 com.croftsoft.apps.chat.view.ChatPanel
11 2003-06-18 com.croftsoft.apps.chat.view.ChatSynchronizer
02 2003-08-02 com.croftsoft.apps.collection.CroftSoftCollection
08 2003-05-10 com.croftsoft.apps.mars.ai.DefaultTankOperator
08 2003-05-12 com.croftsoft.apps.mars.ai.PlayerTankOperator
08 2003-04-29 com.croftsoft.apps.mars.ai.StateSpaceNode
08 2003-05-10 com.croftsoft.apps.mars.ai.TankCartographer
08 2003-04-29 com.croftsoft.apps.mars.ai.TankConsole
08 2003-04-30 com.croftsoft.apps.mars.ai.TankOperator
07 2003-04-17 com.croftsoft.apps.mars.controller.TankController
```



```

07 2003-04-30 com.croftsoft.apps.mars.Main
07 2003-04-14 com.croftsoft.apps.mars.model.AmmoDump
07 2003-04-17 com.croftsoft.apps.mars.model.AmmoDumpAccessor
07 2003-04-03 com.croftsoft.apps.mars.model.Damageable
07 2003-04-17 com.croftsoft.apps.mars.model.Game
07 2003-04-30 com.croftsoft.apps.mars.model.GameAccessor
07 2003-04-13 com.croftsoft.apps.mars.model.Impassable
07 2003-04-14 com.croftsoft.apps.mars.model.Model
07 2003-04-16 com.croftsoft.apps.mars.model.ModelAccessor
07 2003-05-12 com.croftsoft.apps.mars.model.seri.SeriAmmoDump
07 2003-09-10 com.croftsoft.apps.mars.model.seri.SeriGame
07 2003-04-16 com.croftsoft.apps.mars.model.seri.SeriModel
07 2003-05-11 com.croftsoft.apps.mars.model.seri.SeriWorld
07 2003-05-11 com.croftsoft.apps.mars.model.World
07 2003-04-17 com.croftsoft.apps.mars.model.WorldAccessor
10 2003-05-31 com.croftsoft.apps.mars.net.GameData
10 2003-06-12 com.croftsoft.apps.mars.net.GameInit
10 2003-06-13 com.croftsoft.apps.mars.net.MarsServer
10 2003-06-13 com.croftsoft.apps.mars.net.MarsServlet
10 2003-05-13 com.croftsoft.apps.mars.net.NetController
10 2003-09-10 com.croftsoft.apps.mars.net.NetGame
10 2003-06-12 com.croftsoft.apps.mars.net.NetMain
10 2003-06-13 com.croftsoft.apps.mars.net.Player
10 2003-05-13 com.croftsoft.apps.mars.net.request.AbstractRequest
10 2003-05-13 com.croftsoft.apps.mars.net.request.FireRequest
10 2003-05-13 com.croftsoft.apps.mars.net.request.MoveRequest
10 2003-05-13 com.croftsoft.apps.mars.net.request.Request
10 2003-05-13 com.croftsoft.apps.mars.net.request.ViewRequest
10 2003-06-02 com.croftsoft.apps.mars.net.Synchronizer
07 2003-04-17 com.croftsoft.apps.mars.view.AmmoDumpAnimator
07 2003-07-17 com.croftsoft.apps.mars.view.GameAnimator
07 2003-04-17 com.croftsoft.apps.mars.view.ModelAnimator
07 2003-04-17 com.croftsoft.apps.mars.view.WorldAnimator
06 2003-08-11 com.croftsoft.apps.tile.Tile
06 2003-08-11 com.croftsoft.apps.tile.TileData
06 2002-11-04 com.croftsoft.apps.wyrm.entity.PcBean
08 2003-05-09 com.croftsoft.core.ai.astar.AStar
08 2003-05-09 com.croftsoft.core.ai.astar.AStarTest
08 2003-04-29 com.croftsoft.core.ai.astar.Cartographer
08 2003-05-10 com.croftsoft.core.ai.astar.GradientCartographer
08 2003-05-10 com.croftsoft.core.ai.astar.GridCartographer
08 2003-05-09 com.croftsoft.core.ai.astar.NodeInfo
08 2003-04-22 com.croftsoft.core.ai.astar.SpaceTester
07 2003-08-01 com.croftsoft.core.animation.AnimatedApplet
03 2003-07-23 com.croftsoft.core.animation.AnimatedComponent
04 2002-03-15 com.croftsoft.core.animation.Clock
03 2003-11-08 com.croftsoft.core.animation.AnimationFactory
07 2003-08-02 com.croftsoft.core.animation.AnimationInit
04 2003-07-12 com.croftsoft.core.animation.animator.CursorAnimator
04 2003-07-07 com.croftsoft.core.animation.animator.FrameRateAnimator
04 2003-07-24 com.croftsoft.core.animation.animator.NullComponentAnimator
04 2003-07-05 com.croftsoft.core.animation.animator.TileAnimator

```

```

07 2003-07-05 com.croftsoft.core.animation.animator.TileAnimator

04 ----- com.croftsoft.core.animation.clock.*
07 2003-09-10 com.croftsoft.core.animation.clock.Timekeeper
03 2002-12-01 com.croftsoft.core.animation.collector.BooleanRepaintCollector
03 2003-09-05 com.croftsoft.core.animation.collector.CoalescingRepaintCollector
03 2002-12-01 com.croftsoft.core.animation.collector.SimpleRepaintCollector
03 2002-12-01 com.croftsoft.core.animation.ComponentAnimator
05 2003-07-23 com.croftsoft.core.animation.component.BufferedAnimatedComponent
03 2002-03-23 com.croftsoft.core.animation.ComponentPainter
03 2002-03-23 com.croftsoft.core.animation.ComponentUpdater
03 2003-11-08 com.croftsoft.core.animation.factory.DefaultAnimationFactory
05 2002-02-18 com.croftsoft.core.animation.icon.ResourceImageIcon
05 2002-03-13 com.croftsoft.core.animation.icon.VolatileImageIcon
11 2003-06-07 com.croftsoft.core.animation.model.ModelId
11 2003-06-07 com.croftsoft.core.animation.model.seri.SeriModelId
04 2003-07-05 com.croftsoft.core.animation.painter.ArrayComponentPainter
04 2003-08-05 com.croftsoft.core.animation.painter.ColorPainter
04 2003-07-11 com.croftsoft.core.animation.painter.NullComponentPainter
04 2003-07-05 com.croftsoft.core.animation.painter.SpacePainter
04 2003-07-11 com.croftsoft.core.animation.painter.TilePainter
03 2002-12-07 com.croftsoft.core.animation.RepaintCollector
04 2003-07-11 com.croftsoft.core.animation.Sprite
04 2003-07-12 com.croftsoft.core.animation.sprite.AbstractSprite
04 2003-07-11 com.croftsoft.core.animation.sprite.IconSprite
04 2003-07-05 com.croftsoft.core.animation.updater.ArrayComponentUpdater
04 2003-07-11 com.croftsoft.core.animation.updater.BounceUpdater
04 2003-07-05 com.croftsoft.core.animation.updater.EdgeScrollUpdater
04 2003-07-11 com.croftsoft.core.animation.updater.IconSequenceUpdater
04 2003-07-11 com.croftsoft.core.animation.updater.NullComponentUpdater
06 2003-03-14 com.croftsoft.core.applet.AppletLib
02 2003-07-17 com.croftsoft.core.awt.image.ImageLib.crop()
05 2003-07-17 com.croftsoft.core.awt.image.ImageLib.loadAutomaticImage()
05 2003-07-17 com.croftsoft.core.awt.image.ImageLib.loadBufferedImage()
10 2003-06-12 com.croftsoft.core.beans.XmlBeanCoder
05 2003-07-24 com.croftsoft.core.gui.BufferStrategyAnimatedComponent
05 2003-07-25 com.croftsoft.core.gui.DisplayModeLib
05 2003-07-23 com.croftsoft.core.gui.FullScreenToggler
05 2003-07-26 com.croftsoft.core.gui.GraphicsDeviceLib
02 2003-08-02 com.croftsoft.core.gui.LifecycleWindowListener
02 2003-08-02 com.croftsoft.core.gui.multi.MultiApplet
02 2003-05-04 com.croftsoft.core.gui.multi.MultiAppletNews
02 2003-05-04 com.croftsoft.core.gui.multi.MultiAppletStub
09 2003-05-13 com.croftsoft.core.io.Encoder
09 2003-05-14 com.croftsoft.core.io.Parser
10 2003-05-28 com.croftsoft.core.io.SerializableCoder
06 2003-06-13 com.croftsoft.core.io.SerializableLib
10 2003-06-13 com.croftsoft.core.io.SerializableLib.copy()
09 2003-04-30 com.croftsoft.core.io.StreamLib.toString()
09 2003-06-02 com.croftsoft.core.io.StringCoder
02 2002-12-22 com.croftsoft.core.jnlp.JnlpLib
02 2001-10-23 com.croftsoft.core.jnlp.JnlpProxy

```



```

02 2002-12-22 com.croftsoft.core.jnlp.JnlpServices
02 2003-08-02 com.croftsoft.core.jnlp.JnlpServicesImpl
06 2003-08-02 com.croftsoft.core.jnlp.JnlpServicesImpl
06 ----- com.croftsoft.core.lang.classloader.*
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Commissionable
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Destroyable
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Initializable
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Lifecycle
02 2003-09-10 com.croftsoft.core.lang.lifecycle.LifecycleLib
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Resumable
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Startable
02 2003-09-10 com.croftsoft.core.lang.lifecycle.Stoppable
02 2001-09-13 com.croftsoft.core.lang.Pair
06 2003-03-12 com.croftsoft.core.lang.Testable
07 2003-05-13 com.croftsoft.core.math.geom.Circle
09 2003-06-05 com.croftsoft.core.net.http.HttpLib.post()
10 2003-06-12 com.croftsoft.core.net.http.msg.HttpMessageClient
10 2003-06-12 com.croftsoft.core.net.http.msg.HttpMessagePoller
09 2003-05-26 com.croftsoft.core.net.http.msg.HttpMessagePusher
10 1998-11-23 com.croftsoft.core.role.Consumer
09 2002-01-27 com.croftsoft.core.role.Server
11 2003-06-17 com.croftsoft.core.security.Authentication
06 2003-03-27 com.croftsoft.core.security.DigestLib
09 2003-06-03 com.croftsoft.core.servlet.HttpGatewayServlet
06 ----- com.croftsoft.core.util.cache.secure.*
11 2003-06-17 com.croftsoft.core.util.id.Id
11 2003-06-17 com.croftsoft.core.util.id.LongId
03 2003-05-22 com.croftsoft.core.util.loop.FixedDelayLoopGovernor
09 2003-05-22 com.croftsoft.core.util.loop.Looper
09 2003-05-22 com.croftsoft.core.util.loop.Loopable
03 2003-05-22 com.croftsoft.core.util.loop.LoopGovernor
03 2003-05-22 com.croftsoft.core.util.loop.SamplerLoopGovernor
03 2003-11-04 com.croftsoft.core.util.loop.WindowedLoopGovernor
09 2003-06-06 com.croftsoft.core.util.queue.ListQueue
09 2003-06-18 com.croftsoft.core.util.queue.Queue
10 2003-05-27 com.croftsoft.core.util.queue.QueuePuller

```



附录 B CVS 简介

很多开放源代码仓库都使用并发版本控制系统(Concurrent Versions System, CVS)——一个广泛使用的开发源代码仓库版本控制系统。它主要由软件的开发人员用来将源代码的更新递增地保存到中心仓库中,保存工作以任何人都可以撤销任意数目的变更以返回到以前的任意版本的这种方法进行的。开发人员通过 Internet 使用一个 CVS 客户机来周期性地将他们对源代码本地工作的备份和远程中心仓库进行同步。

当有两个或多个编程人员同时使用相同的代码库时,使用一个版本控制系统是完全必要的。然而对单独的一人开发人员来说,它也不是一个坏主意,因为它保持了优秀的配置管理习惯。在所有的版本控制系统中,学习 CVS 是非常有用的,因为它好像是最常用的一个。

版本控制不仅仅只针对源代码。它可以和其他任何文件一起使用,但是最好是和手工输入的纯文本文件一起使用。例如 javadoc 文件 overview.html 和 package.html 都应该和源代码一起检入 CVS。但是一般也不必检入 javadoc 的 HTML 文件,因为这些 HTML 文件是从源代码文件中产生的。

B.1 检出代码

尽管已经有了 CVS 的 GUI 客户应用程序,我仍然喜欢使用命令行接口。它使事情变得更加容易,因为无论在 Unix 系统中还是在 Windows 系统中,命令行接口都相同。在 Linux 系统中, CVS 通常都是预安装的。在 Windows 系统中,我喜欢从 CVS Home 的下载页面上¹下载可以使用的 CVS 客户程序。

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/croftsoft checkout croftsoft
```

假设 CVS 就在当前的路径中,可以使用像前面这样的 CVS 命令从 CVS 仓库中下载某一个模块的一个副本,并将其保存到您的工作目录中。记住,这将会创建一个新的子目录。将这个子目录和代码的其余部分分隔开来,就可以很容易地删除和恢复它,而不需要担心会意外地删除了您自己的某些代码。

```
cvs update -d
```

如果希望更新工作目录和全部子目录以合并自最开始的检出以来所有已经被检入这个仓库的变更,在新子目录内输入上面的这个命令就可以了。参数-d 将会使 CVS 创建所有已经被添加到模块中的新子目录。

但是,需要注意,因为这些更新可能会破坏您的构建——如果这个接口发生了改变的话。为此,您可能希望下载这个代码最新的稳定版本作为旧的归档文件而不是使用 CVS。如果遇到了麻烦,可以很容易地切换回以前的版本。但是,使用 CVS 允许使用仓库里所有代码的最新开发版,无论是稳定的还是不稳定的。

¹ <http://www.cvshome.org/downloads.html>

B.2 创建自己的项目

您可能希望使用 CVS 管理自己的 Java 游戏开发项目。这样将可以协调您的代码和远程协作开发人员的代码的更新，而不必要维护您自己的服务器和备份方式。如果您是独自工作，管理和备份您的代码将会是一个很好的方法。刚开始时，这需要您了解创建仓库和 CVS 命令等方面的知识。

如果您的项目是开放源代码的，那么 SourceForge.net 将可以免费保存它。我喜欢 SourceForge.net 的原因之一就是，它给了您使用一个基于 Web 的接口浏览 CVS 仓库的选择。它还集成了一个开放源代码的工具，这个工具被称为 ViewCVS，它会突出源文件的当前版本和所有以前版本之间的不同——diff。图 B-1 给出了一个这样的示例。

在 SourceForge.net Project Administrator 文档或 Karl Fogel 编写的这本 Open Source Development with CVS 书中可以找到建立 CVS 仓库的更为详细的说明。这本书的一些关于技术方面的章节已经发布在 GDL 下，您可以免费地在线阅读。我不打算详细地介绍 CVS 的用法说明，但是将提供一个快速的核对表，其中会包括一些基本的 CVS 命令，并提供我亲自发现的一些很有用的警告。

```
export CVSROOT=croft@cvs.croftsoft.sourceforge.net:/cvsroot/croftsoft
```

在创建自己的仓库的时候，会希望设置 CVSROOT 环境变量。CVSROOT 环境变量指向 CVS 仓库的根目录，这样就不需要以指向 CVS 命令的命令行参数的形式指定它。上面的示例就是我在 Linux 的 .bashrc 文件里使用的 CVSROOT 变量。

```
export EDITOR=/usr/bin/gedit
```

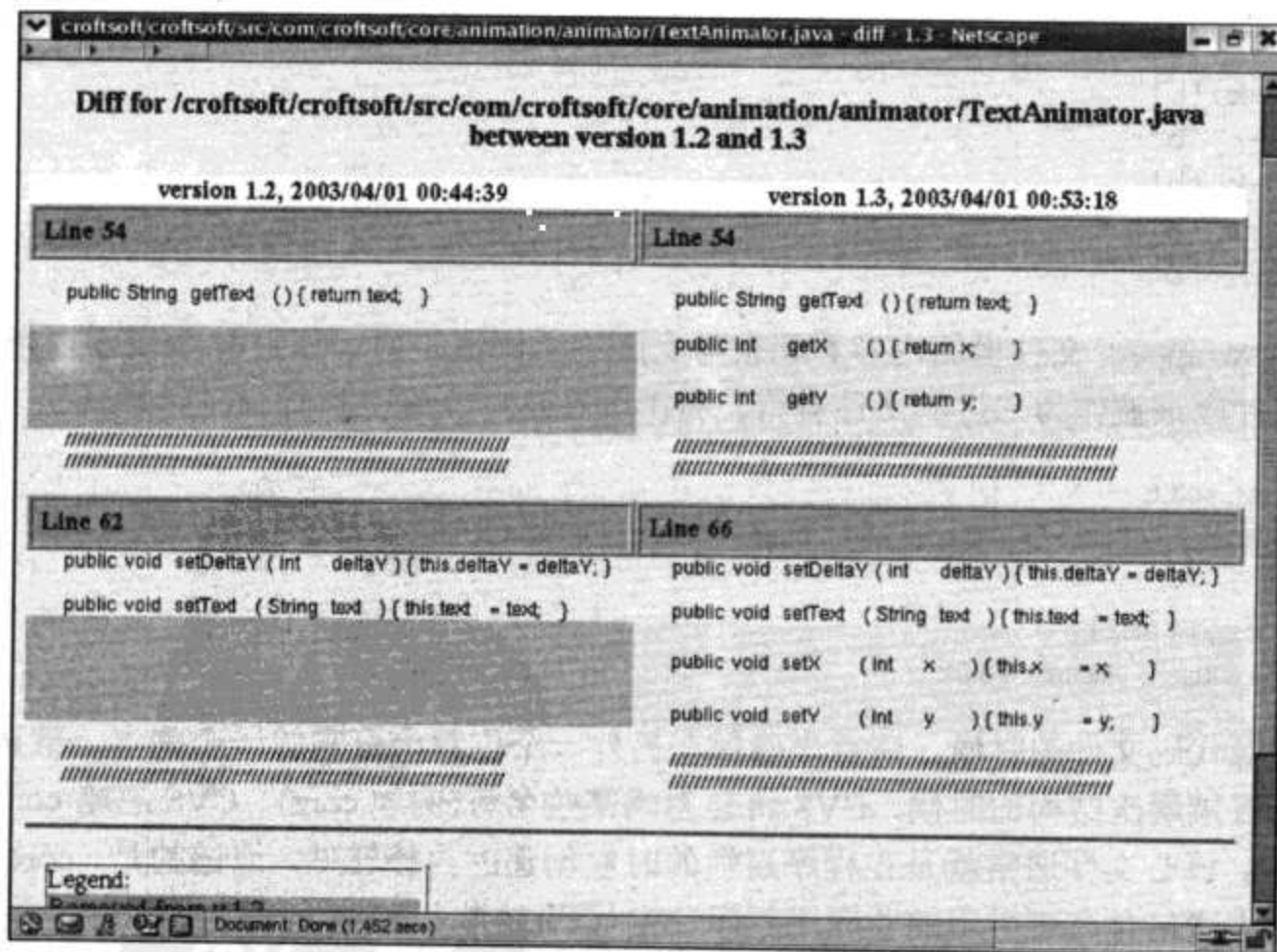


图 B-1 diff

您可能还希望设置 EDITOR 环境变量，因为指定的编辑器将会被 CVS 调用以输入日志消息——每次提交代码变更的时候。我个人对输入这样的消息并不感到麻烦。它们几乎从来都不可读，并且当转移仓库的时候，通常都没有保存它们，这已经成为了我的经验。

```
export CVS_RSH=ssh
```

您也许希望将与服务器的通信进行加密以防止您的密码被别人截获。可以将 CVS_RSH 环境变量设置为表明 CVS 应该使用安全外壳(secure shell, SSH)。SSH 通常在 Linux 系统中是预安装的，但是在 Windows 系统中，您可能还必须想一点办法。

```
cvs checkout CVSROOT
cd CVSROOT
gedit cvswrappers &
cvs commit -m .
```

在源代码文本文件中，CVS 自动地将在 DOS 和 Windows 中使用的回车/换行字符对转换为在 Unix 系统中使用的一个换行字符。由于 CVS 本身不能区分文本和二进制文件，这可能会破坏多媒体文件。为了防止这种事情发生，将需要修改一个称为 CVSROOT 的特殊模块里的 cvswrappers 文件。将需要检出这个模块，编辑这个 cvswrappers 文件，再提交所做的更改。当希望得到一个创建提交日志消息的提示时，就使用 -m 作为传递给 CVS 的 Commit 命令的参数。

```
*.au -k 'b'
*.bmp -k 'b'
*.dat -k 'b'
*.ear -k 'b'
*.ico -k 'b'
*.gif -k 'b'
*.jar -k 'b'
*.jpg -k 'b'
*.mpg -k 'b'
*.png -k 'b'
*.war -k 'b'
*.wav -k 'b'
*.zip -k 'b'
```

我的 cvswrappers 文件里的内容看起来与上面的示例很相似。其中 -k 'b' 表示以指定的扩展名结尾的文件应该被作为二进制文件看待。将不会在这些文件上执行自动行尾转换。

```
cd croftsoft
cvs import -I ! -m "initial import" croftsoft CroftSoft rl
cd ..
rm -r croftsoft
cvs checkout croftsoft
```

在导入源代码文件的时候，请首先确保在另外一个位置上有它的一个副本。默认情况下，在导入一个目录层次结构的时候，CVS 将会忽略某些名称(例如 core)。CVS 忽略 core，因为在 Unix 系统中，核心文件通常就是在程序崩溃的时候创建的内核转储。遗憾的是，core 也是一个我常常用来描述包含高可重用的非应用程序特定代码(这些代码在一个组织内共享)的包目录的一个术语。使用 -I! 选项以强制 import 命令包含所有的目录和文件，而不管它们的名称是什么。


```
cd src/com/croftsoft/apps
mkdir chess
cvs add chess
cvs commit
```

add 命令可以用来向模块添加一个目录(在最初的导入以后)。在上面的示例中, 子包 `com.croftsoft.apps.chess` 被添加到已有的包 `com.croftsoft.apps` 中。

```
cd src/com/croftsoft/core/lang
cvs add *.java
cvs commit
```

也可以使用 CVS 的 **add** 命令来添加某些文件。在上面的这个示例中, 通配符是用来添加这个目录中所有的 Java 源代码文件的。

```
rm ObjectLib.java
cvs remove ObjectLib.java
cvs commit ObjectLib.java
```

为了能够从仓库中删除某一个文件, 必须首先从工作备份中删除这个文件。

```
del ImageLib.java
cvs remove ImageLib.java
cvs commit
cd ..
rmdir image
```

一般情况下, 不应该在没有与其他开发人员协调的情况下从 CVS 仓库中删除一个目录。可以相对安全地从 CVS 仓库中删除这些文件, 再从工作目录中删除这个目录。

```
cvs update DateFormatLib.java
```

也可以使用 **update** 命令更新某一个具体的文件。这在已经决定不提交文件修改的时候非常有用。可以删除修改以后的文件, 并使用 **update** 下载原始版本的一个备份。

B.3 并行程序设计

如果多个开发人员都在修改同一个目录中源代码的时候, 就会存在很多挑战。通常, 由某一个开发人员造成的更改会导致其他开发人员要调试很长时间。版本控制系统不能完全通过它自己解决这个问题。结合开发实践、文档和一些注意事项也都是必要的。

B.3.1 实施代码所有权

当多个开发人员都在同一个库里的源代码上工作的时候, 预防困难的最简单也最安全的方法就是采用代码所有权(`code ownership`)实践。当开发人员在执行代码所有权的时候, 每一个 Java 类的源代码文件都被分配给一个代码所有者(`code owner`), 并且通常将源代码文件按照项目或 Java 包进行分组。当开发人员认识到需要对源代码进行修改或添加某一个东西的时候, 他就和适当的代码所有者进行联系并商量由谁来作这种修改。

在实施代码所有权的时候，开发人员不能在不通知的情况下，将其他开发人员的源代码做一个快速 bug 修复。这其实告诉了代码所有者是谁创建了 bug，或知道是谁创建了 bug，避免了代码所有者发起一个事后的调试会话，修复由其他开发人员按照推测修复的第一个所谓的 bug 时创建的真正 bug，并可以产生一个更具有逻辑性和协调性设计的源代码。

所有修改了源代码的开发人员，无论由于什么原因，都应该将他们的 javadoc 标记 @author 存放在文件里，并更新版本日期。这不仅警告其他开发人员，行为上的改变可能是由于其他开发人员的修改造成的，在需要证书的地方提供证书，也使得代码所有者可以用他们的经验以及他们对某些可能需要调试或增加的代码段的判断捕捉这些问题。后者对继承的源代码和开发人员更换比较多的项目而言非常重要。

如果多个开发人员同时编辑同一个文件，那么提交这种变更的时候，就会导致严重的混乱。人们极力地提倡使用代码所有权防止这种事情发生。

有些版本控制系统，例如 RCS、SCCS 和 Visual SourceSafe，在个别文件层次上自动帮助强制某种形式的代码所有权。默认情况下，开发人员的工作目录里的所有源代码文件都是只读的。当开发人员希望修改这些文件的时候，它就会向中心版本控制服务器发送一个命令。中心服务器再将这些文件标记为未被开发人员修改过的。没有其他开发人员可以意外地同时修改它们各自工作目录中的这个文件。当开发人员完全将所有的变更都提交给仓库的时候，这个文件就在工作目录中被再次标记为只读，这就会将一个命令发送到中心服务器，以便让其他开发人员可以独占式地修改这个文件。

默认情况下，CVS 不支持文件锁定。而是由一个伪智能算法将并发的修改合并起来。这个算法不能处理文件结构里的主要切换，也不能真正地理解内聚源文件代码的语义。我猜想自动合并在过程化编码的时代可能已经很少会引起这些问题，在过程化编码时代，同一个源代码文件的功能往往相互独立。当时，我自己的经验以及我从他人那里听说的经验都是，它可能会严重混乱代码的逻辑，尤其是混乱面向对象代码的逻辑。我真诚地劝告您，为了避免由于缺少人工智能软件合并算法而导致的意想不到的失败，您应该关闭这种特性，并在接收到来自于版本控制系统里潜在的冲突警告消息的时候，手动地合并这种文件。

B.3.2 互相监视

```
cv$ watch on -R croftsoft
```

虽然 VCS 不能防止多个开发人员同时编辑它们各自工作目录里的同一个源代码文件的备份，但是它提供了一个称为 watch 的工具。为了在某一个目录和所有子目录上进行监视，请使用上面的命令，用您的目录名称替换掉 croftsoft 就可以了。

```
cv$ edit Testable.java
cv$ commit Testable.java
cv$ unedit Testable.java
```

通过上面的命令，您也获取了访问某个文件的一些权限。在提交对仓库的更改时，它就会将本地的副本再次置为只读。如果想发布这个文件，并将它再次置为只读而不提交的时候，请使用 unedit 命令。

监视防止了只要发出 CVS 的 edit、commit 或 unedit 命令，就会由发信号的 watchers 通过电子邮件意外地进行同步编辑。一般来说，监视者可以是只要有人准备修改某个给定的源代码

文件，就希望得到警告的任一个开发人员。监视的通告不会发送给引发这个通告事件的人，因此您不会因为自己的编辑而得到警告。

```
cv$ watch add -R core
cv$ watch remove -R core
cv$ editors -R
```

为了将您自己添加为某目录和其所有子目录中文件的一个监视者，请使用 `watch add` 命令。用 `remove` 替换 `add` 将会删除对这些文件的监视。使用 `editors` 命令可以观察任意时刻谁正在修改某一个文件。

B.3.3 创建分支

CVS 允许您将 CVS 仓库的状态的快照打上标签。然后可以使用这个标签创建一个开发分支，该分支独立于对开发主干所做的变更。完成以后，就可以将该分支合并到主干中去。

合并分支可能是非常复杂且很容易产生错误的一个过程。在我看来，为每一个版本创建一个新的项目将会更加简单。例如 `chess1` 的源代码文件就可以复制到一个称为 `chess2` 的新项目中。对旧版本的修补可以较为容易地进行。这使用的磁盘空间可能要比分支多一些，但是它明显地降低了复杂性。硬盘空间通常比人们的时间要便宜。

B.4 参考文献

Fogel, Karl and Moshe Bar. *Open Source Development with CVS, 3rd edition*. Phoenix, AZ: Paraglyph Press, 2003.

